learnvimscriptthehardway

Aha

Published with GitBook



Table of Contents

笨方法学Vimscript面向那些想学会如何自定义Vim编辑器的用户。

这 不是 一个Vim 使用 教程。阅读本书之前你应该适应用Vim编辑文本，并知道一些诸如"buffer", "window"以及"insert mode"这些术语的含义。

本书大致分为三个部分：

- 第一部分讲述了一些Vim的基本命令，通过这些命令修改你的 `~/.vimrc` 可以快速、 方便地定制你的Vim环境。
- 第二部分将Vimscript作为一门编程语言做深入了解，包括变量、比较还有函数。
- 第三部分实例创建一个插件。

本书将一直免费在线阅读。

你可以用8$购买一本电子版(PDF, epub, mobi)。

你可以用20$购买一本纸质版。

你可以用40$购买一本精装版。

可从BitBucket和GitHub获取本书的源码。如果你发下任何错误或感觉某处 可以改善，可以随意提交问题，但是我保留本书的版权，所以，如果你感觉不爽我也能够理解。

原翻译作者为https://github.com/isayme/learnvimscriptthehardway-cn，我只是将其搬到了gitbook。因为有一次原作者的个人网址无法打开，而那时候我还没有找到github的地址，还以为从此失去了一本好书，所以动了将此书搬到github的念头。感谢作者的辛勤工作！

前言

程序猿们更喜欢实现自己的idea。

idea形成算法，而算法又发展为算法，并使 *idea*成真

作为一个程序猿，我们使用文本编辑器记录我们的idea，并为写程序实现它。全职的程序猿一生中的千分之十的时间都是在和他的文本编辑器打交道，这期间他们的所做的事情包括：

- 将他们的灵感记录到计算机上
- 重新考虑并修改灵感中的错误
- 用代码实现他们的灵感
- 写文档记录某功能是如何及为什么那么实现
- 与其他的程序猿交流这个经验

Vim是一个功能相当强大的编辑器，当然，前提是你需要根据你的工作、喜好以及习惯定制它。本书将向你介绍Vimscript，一门用于定制Vim的脚本语言。读完本书你将能够定制 Vim使其更加适应你的文本编辑需求，以后再使用Vim将有更好的体验。

同时我也会提到一些与Vimscript关系不大的点，但那些内容通常都能加强你对Vimscript的认知。 如果你一天仅很少的时间使用Vim，学习Vimscript对你没有多大意义，所以请慎重考虑并平衡你的时间。

本书的写作风格与其他多数的编程书籍略有不同。本书将引领你敲写命令并查看其背后的奥秘， 而不是仅仅简单的告诉你Vimscript是如何工作的。

有时本书会带你进入死胡同，然后才会给你解释解决问题的"正确方法"。其他的书籍通常不这么做， 或者仅仅在解决问题 之后 提到其中的技巧。然而这并不是现实世界中事情的进展顺序。 你时常会快速写一些Vimscript的代码段，运行却遇到不明缘由的故障。细致研读此书，不要 局限于浏览，读完之后再次遇到上述问题你将能够顺利解决了。熟能生巧嘛！

本书的每一章节都只讲述一个主题。每一章节都内容简短而信息丰富，所以不要跳过任何章节。 如果你真想从本书中学到东西，你要动手敲写所有的命令。可能你已经是一个经验丰富的程序猿， 能够快速阅读并理解代码的含义。但是不要掉以轻心，学习Vim/Vimscript有个与其他普通程序 语言更加不同的体验。

你需要 敲写 所有的 命令

你需要 完成 所有的 练习

两个理由解释上述内容的重要性！第一，Vimscript语言是一门古老的语言，其设计难免存在一些 不妥之处。一个简单的配置项就可影响整个脚本的功能。敲写 每个 章节遇到的 每个 命令， 完成 每个 练习，你就会发现很容易地发现并修复遇到的问题。

第二，Vimscript其实 就是 Vim命令。在Vim中，保存一个文件使用命令 `:write` （或者缩写 `:w` ）并回车确认。在Vimscript中，使用 `write` 实现文件保存功能。Vimscript中的许多 命令都可用于日常文件编辑工作，必须勤加练习以记住那些命令才会有用， 仅仅是看过是无法运用自如的。

我希望本书能够对你有所帮助。本书 不是 一本对Vimscript的综合指南。本书试图让你 掌握Vimscript，能够利用它定制你的Vim环境，为其他用户编写一些简单的插件， 能够阅读他人的代码（利用 `:help` 命令），能够分辨出一些常见的语法陷阱。

祝你好运！

鸣谢

首先，我要感谢Zed Shaw帮助我写作[笨方法学Vimscript][]并使之免费。 本书的写作格式及写作风格即受其激发。

同时感谢下列来自Github或Bitbucket的用户：

- aperiodic
- billturner
- chiphogg
- ciwchris
- cwarden
- dmedvinsky
- flatcap
- helixbass
- hoelzro
- jrib
- lheiskan
- lightningdb
- manojkumarm
- manojkumarm
- markscholtz
- marlun
- mattsacks
- Mr-Happy
- mrgrubb
- NagatoPain
- nathanaelkane
- nielsbom
- nvie
- Psycojoker
- riceissa
- rodnaph
- rramsden
- sedm0784
- sherrillmix
- tapichu
- ZyX-I

还有来自更多人的帮助，就不在此一一列举。

预备知识

阅读本书之前，请确保您的机器已经安装了最新版的Vim，本书写作时Vim的最新版本是7.3。新版本的Vim会向后兼容，所以本书中的内容在7.3之后的版本中应该同样有效。

本书中的内容通用，你可以任意选择console Vim或者是gVim、MacVim之类的GUI作为你的终端。

你最好习惯用Vim编辑文件。至少应该知道Vim的基本术语，如"buffer"、"window"、 "normal mode"、"insert mode"、"text mode"。

如果你当前不符合上述的条件，建议你阅读命令 `vimtutor` 的内容、使用Vim一两个月，当你熟练使用Vim后再阅读本书。

你需要有一些编程经验。如果没有，建议先阅读《笨方法学Python 》，读完之后再阅读本书。

> 译者注:原文提到的《笨方法学Python》是英文的，有问题的读者可以选择阅读gastlygem翻译的中文版《笨方法学Python 》

创建Vimrc文件

如果你已经清楚 `~/.vimrc` 的作用并已经有了这个文件，直接跳到下一章继续吧。

`~/.vimrc` 文件包含了Vimscript代码，每次启动Vim时，Vim都会自动执行其中的代码。

在Linux和Mac OS X中，这个文件位于你的home文件夹，并以 `.vimrc` 命名。

在Windows中，这个文件位于你的home文件夹，并以 `_vimrc` 命名。

在任意系统中，在Vim中执行 `:echo $MYVIMRC` 命令可以快速得知这个文件的位置和名称。 文件的路径会在屏幕的底部显示。

如果你的home文件夹没有这个文件，请自行创建一个。

打印信息

Vimscript中，我们最先关注的是 `echo` 和 `echom` 命令。

你可以在Vim中执行 `:help echo` 和 `:help echom` 命令以查看其帮助文档。读完本书之后， 再次遇到新的命令时，你应该先执行 `:help` 命令查看其帮助文档。

执行如下命令，体验 `echo` 命令：

```
:::vim
:echo "Hello, world!"
```

你应该会在屏幕的底部看到 `Hello, world!` 被打印出来。

还是打印消息

现在执行如下命令，体验 `echom` 命令：

```
:::vim
:echom "Hello again, world!"
```

你应该会在屏幕的底部看到 `Hello again, world!` 被打印出来。

执行如下命令，查看上述两个打印命令的区别：

```
:::vim
:messages
```

你应该会看到一些消息。 `Hello, world!` 应该不在其中，但是 `Hello again, world!` 在。

当你写更为复杂的Vimscript时，你可能会想"打印一些信息"以方便调试程序。 `:echo` 命令 会打印输出，但是一旦你的脚本运行完毕，那些输出信息就会消失。使用 `:echom` 打印的信息 会保存下来，你可以执行 `:messages` 命令再次查看那些信息。

注释

继续之前， 咱们先看看如何添加注释。当你写Vimscript脚本时(在你的 `~/.vimrc` 文件中或 其它任意文件)，你可以通过 `"` 字符添加注释，例如：

```
:::vim
" Make space more useful
nnoremap <space> za
```

这个注释方法并不总是有效(这就是Vimscript令人无语的一点)，但是更多的情况这个方法是可以正常工作的。以后我们会谈到什么情况、为什么这个方法会无效。

练习

阅读 `:help echo` 帮助文档。

阅读 `:help echom` 帮助文档。

阅读 `:help messages` 帮助文档。

添加一行代码到你的 `~/.vimrc` 文件中，使得每个打开Vim时都会显示一个可爱的ASCII 字符猫(`>^.^<`)。

设置选项

Vim拥有很多选项可以设置以改变其展现方式。

主要有两种选项：布尔选项（值为"on"或"off"）和键值选项。

布尔选项

执行如下命令：

```
:::vim
:set number
```

如果之前屏幕左侧没有显示行号，那么现在你就会看见行号。执行命令：

```
:::vim
:set nonumber
```

行号应该消失。 `number` 是一个布尔选项：可以off、可以on。通过 `:set number` 命令打开、 `:set nonumber` 命令关闭。

所有的布尔选项都是这种配置方法。 `:set <name>` 打开选项、 `:set no<name>` 关闭选项。

切换布尔选项

你可以"切换"布尔选项的值，即从开启切为关闭或从关闭切为开启。执行命令：

```
:::vim
:set number!
```

行号会再次显示出来。再次执行命令：

```
:::vim
:set number!
```

行号应该会再次消失。添加一个 `!` （感叹号）至布尔选项后面就会切换对于选项的值。

查看选项当前值

你可以使用一个`?`符号向Vim获取一个选项的当前值。执行如下命令并查看每个命令的 返回结果：

```
:::vim
:set number
:set number?
:set nonumber
:set number?
```

注意第一次`:set number?`命令返回的是`number`而第二次返回的是`nonumber`。

键值选项

有些选项并不只有off或on两种状态，它们需要一个值。执行如下命令，查看返回结果：

```
:::vim
:set number
:set numberwidth=10
:set numberwidth=4
:set numberwidth?
```

`numberwidth`选项改变行号的列宽。你可以通过`:set <name>=<value>`命令改变 非布尔选项的选项值，并使用`:set <name>?`命令查看选项的当前值。

来看看一些常用选项的值：

```
:::vim
:set wrap?
:set shiftround?
:set matchtime?
```

一次性设置多个选项

最后，你可以在一个`:set`命令中设置多个选项的值。试试如下命令：

```
:::vim
:set numberwidth=2
:set nonumber
:set number numberwidth=6
```

注意最后一个命令是如何一次性设置两个选项值的。

练习

阅读`:help 'number'`（注意有单引号）帮助文档。

阅读`:help relativenumber`帮助文档。

阅读`:help numberwidth`帮助文档。

阅读`:help wrap`帮助文档。

阅读`:help shiftround`帮助文档。

阅读`:help matchtime`帮助文档。

按你自己的喜好在你的`~/.vimrc`文件中添加几个设置选项。

基本映射

如果说Vimscript有一个特性使得你能够按照你的意愿定制Vim，那就非键盘映射莫属。 你可以通过键盘映射告诉Vim：

> 当我按下这个键时，我需要你放弃默认操作，按我的想法做。

我们先从normal模式的键盘映射开始。我们将在下一章节讨论insert模式和其他模式下的 键盘映射。

随意在文本中敲写几行文字，然后运行命令：

```
:::vim
:map - x
```

将光标置于文本中的某处，按下 `-` 。注意Vim删除了当前光标下的字符，就好像你按了 `x` 一样。

我们本来就有个按键用于 "删除当前光标下的字符" ，所以我们将 `-` 重新映射到稍微有用的 功能。执行命令：

```
:::vim
:map - dd
```

现在移动光标到任意一行，再按下 `-` ，这次Vim删除了整行的文本，因为 `dd` 的功能就是删除整行。

特殊字符

你可以使用 `<keyname>` 告诉Vim一个特殊的按键。尝试如下命令：

```
:::vim
:map <space> viw
```

移动光标到一个单词上，按下空格键。Vim将高亮选中整个单词。

你也可以映射修饰键入Ctrl和Alt。执行：

```
:::vim
:map <c-d> dd
```

现在在键盘上按下 `Ctrl+d` 将执行 `dd` 命令。

注释

还记得我们在第一章讨论的注释么？键盘映射就无法使用注释。尝试如下命令：

```
:::vim
:map <space> viw " Select word
```

现在你再按下空格键，一些恐怖的事情就会发生。想一想为什么会这样呢？

当你按下空格键时，Vim认为你是想执行命令 `viw<space>"<space>Select<space>word` 。 很明显，这不是你的本意。

如果你仔细查看了这个映射的结果，可能你会发现一些奇怪的事。利用几分钟时间， 弄明白使用这个映射时到底发生了什么，以及 为什么 会是那样的结果。

暂时搞不明白也不要担心，我们很快就会再次谈论这个问题。

练习

映射按键 `-` 为 "删除当前行，然后将其粘贴到下一行" 。然后你就可以一次按键就将一行 文本移动到下一行。

将那个映射命令添加到你的 `~/.vimrc` 文件中，以后每次启动Vim你都可以使用那个映射了。

试试如何映射按键 `_` ，使其将当前行上移一行。

将这个映射也加到你的的 `~/.vimrc` 文件中。

想想如果你想取消一个映射或重置一个按键为默认功能时该怎么操作。

模式映射

上一章中我们谈论了如何在Vim中映射按键。我们使用的命令 `map` 在normal模式下工作。 如果阅读本章之前你自己已经折腾了，可能会注意到这个映射在visual模式一样工作。

你可以使用 `nmap` 、 `vmap` 和 `imap` 命令分别指定映射仅在normal、visual、insert模式有效。

执行如下命令：

```
:::vim
:nmap \ dd
```

在normal模式下，按下 `\` 。Vim会删除当前行。

现在进入Visual模式，再次按下 `\` 。什么都不会发生，因为我们告诉了Vim这个映射仅在normal 模式下工作（同时 `\` 的默认行为是什么都不做）。

运行如下命令：

```
:::vim
:vmap \ U
```

进入visual模式并选中一些文字，按下 `\` 。Vim将把选中文本转换成大写格式。

分别在normal模式和visual模式测试 `\` 按键，注意不同模式下的效应。

增强记忆

起初，将同样的按键根据当前模式的不同映射到不同的功能听起来很可怕。为什么每次按下那个键之前都要停下想想我们现在是在什么模式？那样是不是更浪费时间？

实践中我们发现那真不是个问题。一旦你经常使用Vim，你就不会在意你按下的键了。 你会想“删除一行”，而不是“按`dd·”。你的手指和大脑都会记住你的映射，潜意识中你就会 按下那些映射按键。

insert模式

现在我们已经知道如何在normal模式和visual模式下映射按键。现在我们谈谈insert模式下的映射方法。运行如下命令：

```
:::vim
:imap <c-d> dd
```

你可能猜想这个命令的作用是在insert模式下通过按键 `Ctrl+d` 删除整行。这个映射很实用，因为你不必每次都要为了删除某些行而切回到normal模式。

好的我们试一下。它并不如我们想象那般工作，而仅仅是在文件中添加了两个 `d` 字符！ 它压根就没用。

问题就在于Vim只按我们说的做。这个例子中，我们说：“当我按下 `<c-d>` 时，相当于我 按了两次 `d` ”。而当你在insert模式下，按下两次 `d` 的作用就是输入两个字符 `d` 。

要想让这个映射按我们的期望执行，我们需要更加明确的指令。修改映射并运行如下命令：

```
:::vim
:imap <c-d> <esc>dd
```

`<esc>` 告诉Vim按下ESC按键，即退出insert模式。

现在再试试这个映射。它能够正常工作，但是注意你是如何回到normal模式的。这是因为我们 告诉Vim `<c-d>` 要退出insert模式并删除一行，但是我们没有告诉它再回到insert模式。

运行如下命令，修复映射问题：

```
:::vim
:imap <c-d> <esc>ddi
```

结尾的 `i` 告诉Vim进入insert模式，至此我们的映射才最终完成。

练习

设置一个映射，当你在insert模式时，可以通过按下 `<c-u>` 将当前光标所在的单词转换成 大写格式。每次我写一个类似 `MAX_CONNECTIONS_ALLOWED` 这样很长的常量时都能感觉到这个 映射的实用性。因为这样我就可以以小写的格式敲写常量，然后用这个映射将其转成大写， 不必一直需要按着shift键。

将那个映射添加上到你的 `~/.vimrc` 文件中。

设置一个映射，当你在 *normal* 模式时，可以通过按下 `<c-u>` 将当前光标所在的单词转换成大写格式。这个映射和上面那个有点区别，因为你必须要进入normal模式，也不需要结束时切到insert模式。

将那个映射添加上到你的 `~/.vimrc` 文件中。

精确映射

准备好，下面的内容会比较难以理解。

目前为止，我们已经使用`map`、`nmap`、`vmap`以及`imap`创建了实用的按键映射。 他们很方便，但是有个缺点。运行下面的命令：

```
:::vim
:nmap - dd
:nmap \ -
```

试试按下`\`（在normal模式）。有什么现象？

当你按下`\`时，Vim会解释其为`-`。但是我们又映射了`-`！Vim会继续解析`-`为`dd`， 即它会删除整行。

你使用那些命令创建的映射可能会被Vim解释成 其它 的映射。乍一听这像是一个优点， 但实际上这很变态。解释原因之前，我们先用如下命令删除那些映射：

```
:::vim
:nunmap -
:nunmap \
```

递归

运行命令：

```
:::vim
:nmap dd O<esc>jddk
```

上面的命令看上去像是要映射`dd`为：

- 在当前行之前添加新行
- 退出insert模式
- 向下移动一行
- 删除当前行
- 向上移动到新加的行

貌似这个映射的作用是“清除当前行”。但你可以试试。

当你按下 `dd` 后，Vim就不动了。按下 `<c-c>` 才可以继续，但是你的文件中会多出许多 空行！想想发生了什么？

这个映射实际上是 递归 的！当你按下 `dd` 后，Vim解释为：

- `dd` 存在映射，执行映射的内容。
    - 新建一行。
    - 退出insert模式。
    - 向下移动一行。
    - `dd` 存在映射，执行映射的内容。
        - 新建一行。
        - 退出insert模式。
        - 向下移动一行。
        - `dd` 存在映射，执行映射的内容。然后一直这样。

这个映射永远不会结束！删除这个可怕的映射再继续：

```
:::vim
:nunmap dd
```

负面影响

`*map` 系列命令的一个缺点就是存在递归的危险。另外一个是如果你安装一个插件，插件 映射了同一个按键为不同的行为，两者冲突，有一个映射就无效了。

当安装一个新的插件时，可能你不会使用或记住每一个其创建的映射。即使你记住了，你还得 回看下你的 `~/.vimrc` 文件以确保你自定义的映射与插件创建的没有冲突。

这导致插件安装变得乏味，易于出错。肯定有个解决办法。

非递归映射

Vim提供另一组映射命令，这些命令创建的映射在运行时 不会 进行递归。运行命令：

```
:::vim
:nmap x dd
:nnoremap \ x
```

按下 `\` 看看有什么现象。

当你按下 `\` 时，Vim忽略了 `x` 的映射，仅按照 `x` 的默认操作执行。即删除当前光标下的字符而不是删除整行。

每一个 `*map` 系列的命令都有个对应的 `*noremap` 命令，包括： `noremap` / `nnoremap` 、 `vnoremap` 和 `inoremap` 。这些命令将不递归解释映射的内容。

该何时使用这些非递归的映射命令呢？

答案是： 任何时候 。

是的，没开玩笑， 任何时候 。

在安装插件或添加新的自定义映射时使用 `*map` 系列命令纯属是给自己 找 麻烦。 多敲几个字符以确保这个问题不会发生，救自己于火海。

练习

将之前章节中添加到 `~/.vimrc` 文件中的映射命令全部换成非递归版本。

读帮助文档 `:help unmap` 。

Leaders

我们已经学了一种不会让我们发狂的键盘映射方法，但是你可以注意到另外一个问题。

每次我们像 `:nnoremap <space> dd` 这样映射一个按键都会覆盖掉 `<space>` 的原有功能。 如果哪天我们想用 `<space>` 了，怎么办？

有些按键你平常使用并不需要用到。你几乎永远不会用到 `-` 、 `H` 、 `L` 、 `<space>` 、 `<cr>` 和 `<bs>` 这些按键的功能（当然，是在normal模式下）。依据你的工作方式，可能还有其他你不会用到的按键。

这些按键都可以随意映射，但是只有这6个按键貌似不够吧。难道为Vim称道的可定制传说 有问题？

映射按键序列

不像Emacs，Vim可以映射多个按键。运行下面命令：

```
:::vim
:nnoremap -d dd
:nnoremap -c ddO
```

norma模式下快读敲入 `-d` 或 `-c` 查看效果。第一个映射作用是删除一行，第二个是 删除一行并进入insert模式。

这就意味着你可以用一个你不常用的按键（如 `-` ）作为“前缀”，后接其它字符作为一个整体进行映射。你需要多敲一个按键以执行这些映射，多一个按键而已，很容易就记住了。

如果你也认为这是个好方法，我可以告诉你，Vim已经支持这种机制。

Leader

我们称这个“前缀”为“leader”。你可以按你的喜好设置你的leader键。运行命令：

```
:::vim
:let mapleader = "-"
```

你可以替换 `-` 为你喜欢的按键。尽管会屏蔽一个有用的功能，但我个人使用的是 `,` ，因为这个键比较 比较容易按到。

当你创建新的映射时，你可以使用 `<leader>` 表示“我设置的leader按键”。运行命令：

```
:::vim
:nnoremap <leader>d dd
```

现在试试按下你的leader按键和 `d` 。Vim会删除当前行。

然而为何每次都要繁琐的设置 `<leader>` ？为什么创建映射时不直接敲入你的“前缀”按键？ 原因主要有三个。

首先，你某天可能会想要更换你的“leader”。在一个地方定义它使得更方便更换它。

第二，其他人看你的 `~/.vimrc` 文件时，一旦看到 `<leader>` 就能够立即知道你的用意。如果他们 喜欢你的 `~/.vimrc` 配置，即使他们使用不同的leader也可以简单的复制你的映射配置。

最后，许多Vim插件都会创建以 `<leader>` 开头的映射。如果你已经设置了leader，你会更容易上手 使用那些插件。

Local Leader

Vim有另外一个“leader”成为“local leader“。这个leader用于那些只对某类文件 （如Python文件、HTML文件）而设置的映射。

本书将在后续章节讲述如何为特定类型的文件创建映射，但你可以现在创建一个“localleader”：

```
:::vim
:let maplocalleader = "\\"
```

注意我们使用 `\\` 而不是 `\` ，因为 `\` 在Vimscript中是转义字符。我们将在后续章节 讲到这个。

现在你就可以在映射中使用 `<localleader>` 了，使用方法和 `<leader>` 一样（当然， 你要使用另外一个前缀）。

如果你不喜欢反斜线，请随意更改它。

练习

阅读 `:help mapleader` 。

阅读 `:help maplocalleader` 。

在你的 `~/.vimrc` 文件中设置 `mapleader` 和 `maplocalleader` 。

增加 `<leader>` 前缀到之前章节中你添加到 `~/.vimrc` 文件中的映射命令， 防止那些映射覆盖了默认的按键作用。

编辑你的Vimrc文件

在继续学习Vimscript之前，我们先找个添加新映射到 `~/.vimrc` 文件中的更方便的方法。

有时你正在疯狂的编码，突然发现加个映射会加速你的进度。你要立即将其加到 `~/.vimrc` 文件中以防止忘记，但是你 不想 退出当前的文件，因为灵感稍纵即逝。

本章的主题是你想使编辑文件更为方便变得更为方便。

有点绕，但我没有拼错。再读一次。

本章的主题是你想使（（（编辑文件）更为方便）变得更为方便）。

编辑映射

我们在一个分屏中打开 `~/.vimrc` 文件以快速编辑添加映射，然后退出继续编码。运行命令：

```
:::vim
:nnoremap <leader>ev :vsplit $MYVIMRC<cr>
```

我称这个命令为“ 编辑 我的 **vimrc** 文件”。

`$MYVIMRC` 是指定你的 `~/.vimrc` 文件的特殊Vim变量。现在不要担心，相信我不会有问题。

`:vsplit` 打开一个新的纵向分屏。如果你喜好横向的分屏，你可以用 `:split` 替换它。

花一分钟彻底理解理解那个映射命令。命令的目的是：在一个新的分屏中打开我的 `~/.vimrc` 。 它是如何工作的？映射中的每一个字符都是必不可少的？

通过哪个映射，你只要三个键就可以打开你的 `~/.vimrc` 文件。只要你多用几次，你就能 半秒内敲出这个命令。

当你编码过程中突然想到一个可以提高效率的新映射要加到 `~/.vimrc` 文件中， 现在对你来说简直就是小菜一碟。

重读映射配置

`~/.vimrc` 文件添加一个映射并不是立即生效的。 `~/.vimrc` 文件只在你启动Vim的时候才会读取。 也就是说在当前的session中你还需要痛苦的再次拼写那个完整的命令。

我们加个映射来解决这个问题：

```
:::vim
:nnoremap <leader>sv :source $MYVIMRC<cr>
```

我称这个命令为“ 重读 我的 **vimrc** 文件”。

`source` 命令告诉Vim读取指定的文件，并将其当做Vimscript执行。

现在在编码时你可以方便的添加新映射了。

- `<leader>ev` 打开配置文件。
- 添加映射。
- 使用 `:wq<cr>` 或 `ZZ` 保存文件并关闭分屏，回到之前的文件。
- 使用 `<leader>sv` 重读配置使修改生效。

定义一个映射需要8次按键。减少了中断思维的可能性。

练习

在你的 `~/.vimrc` 文件中添加映射，温习“编辑 `~/.vimrc` ”和“重读 `~/.vimrc` ”过程。

多练几遍，随意加些没意义的映射。

阅读 `:help myvimrc` 。

Abbreviations

Vim有个称为"abbreviations"的特性，与映射有点类似，但是它用于insert、replace和command模式。这个特性灵活且强大，不过本节只会谈及最常用的用法。

本书只会讲述insert模式下的abbreviations。运行如下命令：

```
:::vim
:iabbrev adn and
```

进入insert模式并输入：

```
:::text
One adn two.
```

在输入 `adn` 之后输入空格键，Vim会将其替换为 `and` 。

诸如这样的输入纠错是abbreviations的一个很实用的用法。运行命令：

```
:::vim
:iabbrev waht what
:iabbrev tehn then
```

再次进入insert模式并输入：

```
:::text
Well, I don't know waht we should do tehn.
```

注意 两个 abbreviations的替换时机，第二个没有输入空格却也替换了。

Keyword Characters

紧跟一个abbreviation输入"non-keyword character"后Vim会替换那个abbreviation。 "non-keyword character"指那些不在 `iskeyword` 选项中的字符。运行命令：

```
:::vim
:set iskeyword?
```

你将看到类似于 `iskeyword=@,48-57,_,192-255` 的结果。这个格式很复杂，但本质上 "keyword characters"包含一下几种：

- 下划线字符 (`_`).
- 所有字母字符，包括大小写。
- ASCII值在48到57之间的字符（数字0-9）。
- ASCII值在192到255之间的字符（一些特殊ASCII字符）。

如果你想阅读这个选项格式的 完整 描述，你可以运行命令 `:help isfname` ， 另外 阅读之前最好准备点吃的。

你只要记住输入非字母、数字、下划线的字符就会引发abbreviations替换。

更多关于abbreviations

Abbreviations不仅仅只能纠错笔误。我们可以加几个日常编辑中常用的abbreviations。 运行如下命令：

```
:::vim
:iabbrev @@    steve@stevelosh.com
:iabbrev ccopy Copyright 2013 Steve Losh, all rights reserved.
```

随意更换我的名字和邮箱地址为你的，然后试试这两个abbreviations吧~

这些abbreviations将你常用的一长串字符压缩至几个字符，省的每次都要那么麻烦。

Why Not Use Mappings?

为什么不用Mappings?

不错，abbreviations和mappings很像，但是他们的定位不同。看个例子：

运行命令：

```
:::vim
:inoremap ssig -- <cr>Steve Losh<cr>steve@stevelosh.com
```

这个 *mapping* 用于快速插入你的签名。进入insert模式并输入 `ssig` 试试看。

看起来一切正常，但是还有个问题。进入insert模式并输入如下文字：

```
:::text
Larry Lessig wrote the book "Remix".
```

注意到Vim将Larry名字中的 `ssig` 也替换了！mappings不管被映射字符串的前后字符是什么--它只在文本中查找指定的字符串并替换他们。

运行下面的命令删除上面的mappings并用一个abbreviation替换它：

```
:::vim
:iunmap ssig
:iabbrev ssig -- <cr>Steve Losh<cr>steve@stevelosh.com
```

再次试试这个abbreviation。

这次Vim会注意 `ssig` 的前后字符，只会在需要的时候替换它。

Exercises

在你的 `~/.vimrc` 文件中为经常拼写错误的单词增加abbreviations配置。一定要使用 上一章中你创建的mappings来重新打开读取 `~/.vimrc` 文件。

为你的邮箱地址、博客网址、 签名添加abbreviations配置。

为你经常输入的文本添加abbreviations配置。

更多的Mappings

迄今为止我们已经说了很多mappings的内容，但现在我们要再次实践一下。mappings是 使得Vim编辑更为高效的方便快捷途径之一，有必要多加用心。

有个概念在多个例子中出现过，但是我们都没有明确解释，那就是多字符mappings的连续性。

运行如下命令：

```
:::vim
:nnoremap jk dd
```

确保你出于normal模式，快速输入 `jk` 。Vim会删除当前行。

现在试试先输入 `j` ，停顿一下。如果你输入 `j` 后没有快速输入 `k` ，Vim就会判定你不想 生效那个映射，而是将 `j` 按默认操作运行（下移一行）。

这个映射会给光标移动操作带来麻烦，我们先删除它。运行下面的命令：

```
:::vim
:nunmap jk
```

现在normal模式下快速输入 `jk` 会像往常一样下移一行然后又上移一行。

一个更为复杂的Mapping

你已经见过很多简单的mappings了，是时候看看一些复杂的了。运行下面的命令：

```
:::vim
:nnoremap <leader>" viw<esc>a"<esc>hbi"<esc>lel
```

那是一个有趣的mappings！你自己可以先试试。进入normal模式，移动光标至一个单词， 输入 `<leader>"` 。Vim将那个单词用双引号包围！

它是如何工作的呢？我们拆分这个映射并逐个解释：

```
:::text
viw<esc>a"<esc>hbi"<esc>lel
```

- `viw`：高亮选中单词
- `<esc>`：退出visual模式，此时光标会在单词的最后一个字符上
- `a`：移动光标至当前位置之 后 并进入insert模式
- `"`：插入一个 `"`
- `<esc>`：返回到normal模式
- `h`：左移一个字符
- `b`：移动光标至单词头部
- `i`：移动光标至当前位置之 前 并进入insert模式
- `"`：插入一个 `"`
- `<esc>`：返回到normal模式
- `l`：右移一个字符，光标置于单词的头部
- `e`：移动光标至单词尾部
- `l`：右移一个字符，置光标位置在第一个添加的引号上

记住：因为我们使用的是 `nnoremap` 而不是 `nmap` ，所以尽管你映射了字符序列中的字符 也不会有影响。Vim会将其中的字符按默认功能执行。

希望你能看出Vim mappings的潜能及随之引发的阅读困难。

Exercises

像刚才一样创建一个mapping，用单引号而不是双引号。

试试用 `vnoremap` 添加一个mapping，使其能够用引号将你 高亮选中 的文本包裹。 你可能会需要 `` `< `` 和 `` `> `` 命令，所以先执行 `` :help `< `` 看看帮助文档。

将normal模式下的 `H` 映射为移动到当前行的首部。 `h` 是左移，所以你可以认为 `H` 是 “加强版”的 `h` 、

将normal模式下的 `L` 映射为移动到当前行的尾部。 `l` 是右移，所以你可以认为 `L` 是 “加强版”的 `l` 、

读取帮助文档 `:help H` 和 `:help L` 看看你覆盖了哪些命令。考虑考虑这会不会影响你。

将这些mappings添加到你的 `~/.vimrc` 文件中，确保用你的“编辑 `~/.vimrc` ”和“重读 `~/.vimrc` ” 映射操作~

锻炼你的手指

这一章我们会讲到怎么更有效地学习Vim，不过在此之前需要先做一些小小的准备。

让我们先创建一个mapping，这个mapping会为你的左手减轻很多负担。执行下面的命令：

```
:::vim
:inoremap jk <esc>
```

ok，现在进入插入模式然后敲击 `jk` 。Vim会返回到常用模式，就像你敲击了escape按键一样。

在Vim中有很多默认的方式可以退出插入模式：

- `<esc>`
- `<c-c>`
- `<c-[>`

使用上面的这几种方式，你都需要伸出你的爪子，这会让你感到很不自在。使用 `jk` 会很棒，因为这两个按键正好就在你最强健有力的两个手指下面，并且你不用搞得好像在演奏吉他和弦似的移动手指。

有些人可能更喜欢使用 `jj` ，但我更钟意 `jk` ，有两个原因：

- 使用两个不同的按键，你可以“滚动”你的手指而不是把同一个按键按两次。
- 如果你出于习惯在常用模式下按了 `jk` ，只会将光标向下移动一下，然后又向上移动一下，最终光标还是停留在原来的位置。但是在常用模式下按了 `jj` 的话，只会把光标移动到一个不同的地方。

不过需要注意的是，如果在你所用的语言中 `jk` 会经常组合出现（例如德语）,你可能就需要选择一个不同的mapping了。

学习Map

ok，现在你已经有了一个新的mapping，你会怎么学习使用它呢？特别是你已经用了这么久的escape按键，使用escape按键都已经刻入了你的脑子中，以至于在编辑的时候你会不假思索的敲击它。

重新学习一个mapping的窍门就是强制将之前的按键设置为不可用，强迫自己使用新的mapping。执行下面的命令：

```
:::vim
:inoremap <esc> <nop>
```

这个命令会告诉Vim在插入模式下敲击escape按键后执行 `<nop>` (no operation)，这样escape按键在插入模式下就无效了。ok，现在你就不得不使用 `jk` 这个mapping来退出插入模式了。

一开始你可能不太适应，你还是会在插入模式下敲击escape按键，并且以为已经退出到了常用模式，然后开始敲击按键准备在常用模式下做一些事情，从而导致一些不需要的字符出现在你的文本中。这会让你感到有些不爽，但如果你坚持一段时间后，你会惊讶的发现你的思维和手指会多么快的适应 新的mapping。用不到一到两个小时你就不会再在插入模式下敲击escape了。

这个方法适用于所有的用于替代原有操作方式的新mapping，包括在生活中也是如此。当你想改掉一个坏习惯的时候，你最好能够想一些办法使得这个坏习惯很难甚至是不能发生。

如果你想学会自己做饭，不想每天都吃盖浇饭，那么每天下班的时候就不要去成都小吃了。这样你就会在饿了的时候想办法去做点东西吃，当前你先要确保你家里没有泡面。

如果你想戒烟，那你就不要把烟带在身上，把它放到车上。这样当你的烟瘾又犯了的时候，你会觉得走到车里去取烟是一件很蛋疼的事，这样你就不会吸了。

练习

如果还是会在Vim的常用模式中使用方向键移动光标，那么就将它们映射为 `<nop>` 。

如果你在编辑模式下也会使用方向键，同样的，映射为 `<nop>` 吧。

正确使用Vim的关键就是使得自己能够快速的离开插入模式，然后在常用模式下进行移动。

本地缓冲区的选项设置和映射

现在我们先花点时间复习一下我们已经谈论过的三个东西：映射（mappings），缩写（abbreviations）和选项设置（options），这个过程中会讲到一些新的东西。我们将在一个单一的缓冲区中同时设置它们。

这一章所讲到的东西会在下一章中真正的显示它们的作用，目前我们只需先打下基础。

在这一章中你需要在Vim中打开两个文件，两个文件是分开的。我先将它们命名为 `foo` 和 `bar` ，你可以随便对它们命名。然后为每个文件输入一些文字。

映射

选择文件 `foo` ，然后执行下面的命令：

```
:::vim
:nnoremap          <leader>d dd
:nnoremap <buffer> <leader>x dd
```

现在保持在文件 `foo` 下面，确保当前处于常用模式下，然后敲击 `<leader>d` 。Vim会删除一行。这个之前讲到过，没什么新鲜的。

仍然保持在文件 `foo` 下面，敲击 `<leader>x` 。Vim也会删除一行。这很正常，因为我们也将 `<leader>x` 映射到 `dd` 了。

现在切换到文件 `bar` 。在常用模式下敲击 `<leader>d` 。同样的，Vim会删除当前行，也没有什么新鲜的。

ok，现在来点新鲜的：在文件 `bar` 下敲击 `<leader>x` 。

Vim只删除了一个字符，而不是删除整个行！ 为什么会这样？

第二个 `nnoremap` 命令中的 `<buffer>` 告诉Vim这个映射只在定义它的那个缓冲区中有效：

当你在 `bar` 文件下敲击 `<leader>x` ，Vim找不到一个跟它匹配的映射，它将会被解析了两个命令： `<leader>` （这个什么都不会干）和 `x` （通常会删除一个字符）。

本地Leader

在这个例子中， `<leader>x` 是一个本地缓冲区映射，不过这种定义方式并不合适。如果我们需要设定一个只会用于特定缓冲区的映射，一般会使用 `<localleader>` ，而不是 `<leader>` 。

使用两种不同的leader按键就像设置了一种命名空间，这会帮助你保证所有不同的映射对你而言更加清晰直接。

但你在编写一个会被其他人用到的插件的时候，这点显得尤其重要。使用 `<localleader>` 来设置本地映射会防止你的插件覆盖别人用 `<leader>` 设置的全局映射，因为他们可能已经对他们做设置的全局映射非常之习惯了。

设置

在这本书的前面几个章节里，我们谈论了使用 `set` 来设置选项。有一些选项总是会适用于整个Vim，但是有些选项可以基于缓冲区进行设置。

切回到文件 `foo` ，执行下面的命令：

```
:::vim
:setlocal wrap
```

然后切换到文件 `bar` ，执行下面的命令：

```
:::vim
:setlocal nowrap
```

把你的Vim窗口调小一些，你会发现有些行在 `foo` 中会自动换行，而在 `bar` 中则不会。

让我们来测试下另外一个选项。切换到 `foo` 执行下面的命令：

```
:::vim
:setlocal number
```

现在切换到 `bar` ，然后执行下面的命令：

```
:::vim
:setlocal nonumber
```

现在在文件 `foo` 中会出现行号，而在 `bar` 则没有。

不是所有的选项都可以使用 `setlocal` 进行设置。如果你想知道某个特定的选项是否可以设置为本地选项，执行 `:help` 查看它的帮助文档。

对于本地选项如何真正地地工作，我说的有些简略。在练习中你会学到更多这方面的细节。

遮盖

ok，在开始下一节之前，我们先来看关于本地映射的一个非常有趣的特性。切换到文件 `foo` ，然后执行下面的命令：

```
:::vim
:nnoremap <buffer> Q x
:nnoremap          Q dd
```

然后敲击 `Q` ，看看会发生什么？

当你敲击 `Q` ，Vim会执行第一个映射，而不是第二个，因为第一个映射比起第二个要显得更具体，这可以看成第二个映射被第一个映射遮盖了。

切换回文件 `bar` ，然后敲击 `Q` ，Vim会使用第二个映射。这是因为在这个缓冲区中第二个映射没有被第一个映射遮盖。

练习

阅读 `:help local-options` 。

阅读 `:help setlocal` 。

阅读 `:help map-local` 。

自动命令

现在我们谈论一个跟映射一样重要的东西：自动命令。

自动命令可以让Vim自动执行某些指定的命令，这些指定的命令会在某些事件发生的时候执行。我们先看一个例子。

使用 `:edit foo` 打开一个新文件，然后立即使用 `:quit` 关闭。查看你的硬盘，你会发现这个文件并不存在。这是因为在你第一次保存这个文件之前，Vim实际上并没有真正创建它。

让我们对Vim做一些改变，使得Vim可以在你开始编辑文件的时候就创建它们。执行下面的命令：

```
:::vim
:autocmd BufNewFile * :write
```

这里面有很多需要进一步说明的，不过在此之前我建议你先感受下它是怎么工作的。执行 `:edit foo` ，使用 `:quit` 关闭，然后查看硬盘。这个时候文件会存在（当然文件内容为空）。

你只有关闭Vim才能删除这个自动命令。我们会在后面的章节说明如何避免这种情况。

自动命令结构

让我们来深入分析下我们刚才创建的自动命令：

```
:::text
:autocmd BufNewFile * :write
         ^          ^ ^
         |          | |
         |          | 要执行的命令
         |          |
         |          用于事件过滤的“模式（pattern）”
         |
         要监听的“事件”
```

这个命令的第一部分是我们想监听的事件的类型。Vim提供了很多可以监听的事件。这些事件包括：

- 开始编辑一个当前并不存在的文件。
- 读取一个文件，不管这个文件是否存在。
- 改变一个缓冲区的 `filetype` 设置。

- 在某段时间内不按下键盘上面的某个按键。
- 进入插入模式。
- 退出插入模式。

上面只举出了可用事件里面的很小一部分。还有很多其他的事件，你可以利用这些事件来做一些有趣的事情。

这个自动命令的下一部分是一个“模式”，这个模式可以进一步限定你要执行的命令的执行范围。新开一个Vim实例，执行下面的命令：

```
:::vim
:autocmd BufNewFile *.txt :write
```

这个跟之前的那个自动命令基本一样，不过这个自动命令只对后缀为 `.txt` 的文件有效，也就是说当你新建的文件为txt文件的时候，Vim会在文件创建的时候自动执行write命令将文件保存到硬盘上。

试试执行 `:edit bar` ，然后执行 `:quit` ，再执行 `:edit bar.txt` ，然后再执行 `:quit` 。你会发现Vim会自动创建 `bar.txt` ，但不会创建 `bar` ，因为它的后缀名不是txt，不跟模式匹配。

这个自动命令的最后一部分是事件发生时我们想执行的命令。这个部分很容易理解，跟我们执行其他命令一样，除了不能在这个命令中使用特殊的字符，例如 `<cr>` 。我们会在本书后面的章节中谈论如何突破这个限制，现在你只需要遵守它就可以。

再来一个示例

我们再定义一个自动命令，这次使用一个不同的事件。执行下面的命令：

```
:::vim
:autocmd BufWritePre *.html :normal gg=G
```

这里用到了 `normal` 命令，我会在本书的后面的章节里面讲到它，这可能有点超前，不过我觉得这是一个很好的使用自动命令的示例，所以请大家先忍受一下。

创建一个名为 `foo.html` 的新文件。用Vim编辑它，并输入下面的文本，请保证输入的文本完全一致，包括空白符：

```
:::html
<html>
<body>
 <p>Hello!</p>
                 </body>
                  </html>
```

执行 `:w` 保存这个文件。看看会发生了什么？Vim似乎在文件保存之前重新进行了文本缩进处理。

ok，请先相信我文本缩进处理是 `:normal gg=G` 干的，先别纠结于为什么 `:normal gg=G` 可以干这个。

我们应该把注意力放在自动命令上。这个自动命令里面用到的事件是 `BufWritePre` ，这个事件会在你保存任何字符到文件之前触发。

我们使用了 `*.html` 这个模式，这个模式会保证命令只会在编辑html文件的时候被执行。这就是自动命令强大的地方，因为它可以专门针对特定类型的文件来执行我们想要执行的命令。ok，让我们继续探索它吧。

多个事件

你可以创建一个绑定多个事件的自动命令，这些事件使用逗号分隔开。执行下面的命令：

```
:::vim
:autocmd BufWritePre,BufRead *.html :normal gg=G
```

这个跟上面的自动命令基本一样，不同的是它会让Vim不仅在写html文件的时候进行缩进处理，读html文件的时候也会进行缩进处理。如果你有些同事不喜欢把HTML文件格式搞得漂亮点，那么这个命令会很有用。

在Vim脚本编程中有一个不成文的规定，你应该同时使用 `BufRead` 和 `BufNewFile` （译注：这里不是BufWritePre）这两个事件来运行命令，这样当你打开某个类型的文件，不论这个文件是否存在命令都会执行。执行下面的命令：

```
:::vim
:autocmd BufNewFile,BufRead *.html setlocal nowrap
```

上面的命令会使得无论你在什么时候编辑HTML文件自动换行都会被关闭。

FileType事件

最有用的事件是 `FileType` 事件。这个事件会在Vim设置一个缓冲区的 `filetype` 的时候触发。

让我们针对不同文件类型设置一些有用的映射。运行命令：

```
:::vim
:autocmd FileType javascript nnoremap <buffer> <localleader>c I//<esc>
:autocmd FileType python     nnoremap <buffer> <localleader>c I#<esc>
```

打开一个Javascript文件（后缀为 `.js` 的文件）,将光标移动到某一行，敲击 `<localleader>c` ，光标所在的那一行会被注释掉。

现在打开一个Python文件（后缀为 `.py` 的文件）,将光标移动到某一行，敲击 `<localleader>c` ，同样的那一行会被注释掉，不同的是此时所用的是Python的注释字符！

在自动命令中包含我们上一章中学到的本地缓冲区映射，我们可以创建一些映射，这些映射会根据我们正在编辑的文件的类型来进行不同的处理。

这可以为我们在编码的时候减轻很多思考的负担。如果要添加一个注释，我们可能想到的是必须将光标移动到行首，然后添加一个注释字符，而使用上面的映射，我们只需要简单的将其理解为“注释掉这一行”。

练习

浏览 `:help autocmd-events` 查看自动命令可以绑定的所有事件。你不需要现在就记住每一个事件。仅仅只需要了解下你可以使用这些事件做哪些事情。

创建一些 `FileType` 自动命令使用 `setlocal` 对你喜欢的文件类型做一些设置。你可以针对不同的文件类型设置 `wrap` 、 `list` 、 `spell` 和 `number` 这些选项。

对一些你会经常处理的文件类型创建一些类似“注释掉这一行”的命令。

把所有这些自动命令写到你的 `~/.vimrc` 文件里面。记住使用前面章节中提到过的快速编辑和加载 `~/.vimrc` 文件的映射来做这个事情，这是必须的！

本地缓冲区缩写

上一章讲的东西比较多，完全理解会有点难，所以这一章来点容易的。我们已经学习了如何定义本地缓冲区的映射和设置选项，现在以同样的方式来学习本地缓冲区的缩写。

打开你的 `foo` 和 `bar` 这两个文件，切换到 `foo` ，然后执行下面的命令：

```
:::vim
:iabbrev <buffer> --- &mdash;
```

在文件 `foo` 下进入插入模式输入下面的文本：

```
:::text
Hello --- world.
```

Vim会为你将 `---` 替换为“Hello“。现在切换到 `bar` 试试。在 `bar` 中替换不会发生，这是因为我们所定义的缩写被设置为只用于 `foo` 的本地缓冲区。

自动命令和缩写

使用本地缓冲区的缩写和自动命令来创建一个简单的“snippet”系统。

执行下面的命令：

```
:::vim
:autocmd FileType python     :iabbrev <buffer> iff if:<left>
:autocmd FileType javascript :iabbrev <buffer> iff if ()<left>
```

打开一个Javascript文件然后输入 `iff` 缩写。然后再打开一个Python文件试试。Vim会依据文件类型在当前行执行合适的缩写。

练习

为你经常编辑的文件创建更多的针对不同类型的文件的“snippet”缩写。你可以为绝大多数语言创建 `return` 的缩写，为javascript创建 `function` 的缩写，以及为HTML文件创建 `“` 和 `”` 的缩写。

将你创建的snippets加入到你的 `~/.vimrc` 文件中。

记住：最好的学习使用这些snippets的方法是禁用之前你做这些事情的方式。执行 `:iabbrev <buffer> return NOPENOPENOPE` 会强迫你使用缩写，这个命令在你输入return的时候不会输出任何东西。为了节省学习的时间，为你刚才创建的snippets都创建一个上面的缩写来强迫你使用你创建的snippets。

自动命令组

前面几章我们学习了自动命令。执行下面命令：

```
:::vim
:autocmd BufWrite * :echom "Writing buffer!"
```

现在使用`:write`命令将当前缓冲区写入文件，然后执行`:messages`命令查看消息日志。你会看到`Writing buffer!`在消息列表中。

然后将当前缓冲区写入文件，执行`:messages`查看消息日志。你会看到`Writing buffer!`在消息列表中出现了两次。

现在再次执行上面的自动命令：

```
:::vim
:autocmd BufWrite * :echom "Writing buffer!"
```

再次将当前缓冲区写入文件并执行`:messages`命令。你会看到`Writing buffer!`在消息列表中出现了*4*次，这是怎么回事？

这是因为当你以上面的方式创建第二个自动命令的时候，Vim没办法知道你是想替换第一个自动命令。在上面的示例中，Vim创建了两个不同的自动命令，并且这两个命令刚好做同样的事情。

这会有什么问题？

既然你现在知道了Vim可能创建两个完全一样的自动命令，你可能会想："有什么大不了？只要别这么干就可以！"。

问题是当你加载你的`~/.vimrc`文件的时候，Vim会重新读取整个文件，包括你所定义的任何自动命令！这就意味着每次你加载你的`~/.vimrc`文件的时候，Vim都会复制之前的自动命令，这会降低Vim的运行速度，因为它会一次又一次地执行相同的命令。

你可以执行下面的命令模拟这种情况：

```
:::vim
:autocmd BufWrite * :sleep 200m
```

现在将当前缓冲区写入文件。你可能注意到Vim在写入文件的时候有点缓慢，当然也你可能注意不到。现在执行上面的自动命令三次：

```
:::vim
:autocmd BufWrite * :sleep 200m
:autocmd BufWrite * :sleep 200m
:autocmd BufWrite * :sleep 200m
```

再次写文件。这次会更明显。

当然你不会创建任何只是进行sleep而不做任何事情的自动命令，不过一个使用Vim的老鸟的 `~/.vimrc` 文件可以轻易达到1000行，其中会有很多自动命令。再加上安装的插件中的自动命令，这肯定会影响Vim的速度。

把自动命令放到组中（Grouping Autocommands）

对于这个问题，Vim有一个解决方案。这个解决方案的第一步是将相关的自动命令收集起来放到一个已命名的组（groups）中。

新开一个Vim实例，这样可以清除之前所创建的自动命令。然后运行下面的命令：

```
:::vim
:augroup testgroup
:    autocmd BufWrite * :echom "Foo"
:    autocmd BufWrite * :echom "Bar"
:augroup END
```

中间两行的缩进没有什么含义，如果你不想输入的话可以不输。

将一个缓冲区写入文件然后执行 `:messages` 。你应该可以在消息日志列表中看到 `Foo` 和 `Bar` 。现在执行下面的命令：

```
:::vim
:augroup testgroup
:    autocmd BufWrite * :echom "Baz"
:augroup END
```

当你再次将缓冲区写入文件的时候猜猜会发生什么。ok，你也许已经有结果了，重新写入缓冲区，然后执行 `:messages` 命令，看看你猜对了没。

清除自动命令组

当你写入文件的时候发生什么了？猜对了么？

如果你认为Vim会替换那个组，那么你猜错了。不要紧，很多人刚开始的时候都会这么想（我也是）。

当你多次使用 `augroup` 的时候，Vim每次都会组合那些组。

如果你想清除一个组，你可以把 `autocmd!` 这个命令包含在组里面。执行下面的命令：

```
:::vim
:augroup testgroup
:    autocmd!
:    autocmd BufWrite * :echom "Cats"
:augroup END
```

现在试试写入文件然后执行 `:messages` 查看消息日志。这次Vim只会输出 `Cats` 在消息列表中。

在Vimrc中使用自动命令

既然我们现在知道了怎么把自动命令放到一个组里面以及怎么清除这些组，我们可以使用这种方式将自动命令添加到 `~/.vimrc` 中，这样每次加载它的时候就不会复制自动命令了。

添加下面的命令到你的 `~/.vimrc` 文件中：

```
:::vim
augroup filetype_html
    autocmd!
    autocmd FileType html nnoremap <buffer> <localleader>f Vatzf
augroup END
```

当进入 `filetype_html` 这个组的时候，我们会立即清除这个组，然后定义一个自动命令，然后退出这个组。当我们再次加载 `~/.vimrc` 文件的时候，清除组命令会阻止Vim添加一个一模一样的自动命令。

练习

查看你的 `~/.vimrc` 文件，然后把所有的自动命令用上面组的方式包裹起来。如果你觉得有必要，可以把多个自动命令放到一个组里面。

想想上一节的示例中的自动命令是干啥的。

阅读 `:help autocmd-groups` 。

Operator-Pending映射

这一章我们将来探索Vim映射系统中另外一个神奇的部分："Operator-Pending映射"。开始之前，我们先解释下这里面的几个词含义。

一个Operator（操作）就是一个命令，你可以在这个命令的后面输入一个Movement（移动）命令，然后Vim开始对文本执行前面的操作命令，这个操作命令会从你当前所在的位置开始执行，一直到这个移动命令会把你带到的位置结束。

常用到的Operator有 `d` ， `y` 和 `c` 。例如：

```
:::text
按键    操作         移动
----    --------    -------------
dw      删除         到下一个单词
ci(     修改         在括号内
yt,     复制         到逗号
```

Movement映射

Vim允许你创建任何新的movements，这些movements可以跟所有命令一起工作。执行下面的命令：

```
:::vim
:onoremap p i(
```

在缓冲区中输入下面的文字：

```
:::text
return person.get_pets(type="cat", fluffy_only=True)
```

把光标放到单词"cat"上，然后敲击 `dp` 。结果会发生什么？Vim会删除括号内的所有文字。你可以把这个新建的movements当作"参数"。

`onoremap` 命令会告诉Vim当它在等待一个要附加在operator后面的movement的时候，如果这个movement是 `p` 的话，它会把它当作 `i(` 。所以当我们在运行 `dp` 的时候，就象是在对Vim说"delete parameters"，而Vim会把它理解为"在括号内删除"。

我们现在可以立马对所有的operators使用这个新建的映射。再次在缓冲区中输入上面的文字（或者直接把之前修改恢复一下）。

```
:::text
return person.get_pets(type="cat", fluffy_only=True)
```

把光标放到单词“cat”上，然后敲击 `cp` 。这次又会发生什么？Vim会删除括号中的所有文字，不过这一次删除之后Vim会处于插入模式，这是因为你使用的是“change”，而不是“delete”。

再看一个示例。执行下面的命令：

```
:::vim
:onoremap b /return<cr>
```

现在把下面的文字输入到缓冲区：

```
:::python
def count(i):
    i += 1
    print i

    return foo
```

把光标放到第二行的 `i` 上，然后按下 `db` 。会发生生么？Vim把整个函数体中直到 `return` 上面的内容都删除了， `return` 就是上面的映射使用Vim的通用查找得到的结果。

当你想搞清楚怎么定义一个新的operator-pending movement的时候，你可以从下面几个步骤来思考：

1. 在光标所在的位置开始。
2. 进入可视模式(charwise)。
3. ... 把映射的按键放到这里 ...
4. 所有你想包含在movement中的文字都会被选中。

你所要做的工作就是在第三步中填上合适的按键。

改变开始位置

你可能已经从上面所学的东西中意识到一个了问题。如果我们定义的movements都是从光标所在的位置开始的话，那么这就会限制我们做一些我们想使用movement来做的事情。

但是Vim并不会限制你去做你想做的事情，所以对于这个问题肯定有解决办法。执行下面的命令：

```
:::vim
:onoremap in( :<c-u>normal! f(vi(<cr>
```

这个命令看起来有些复杂，不过我们还是先试试它能干什么。将下面的文字输入缓冲区：

```
:::python
print foo(bar)
```

把光标放到单词 `print` 上面，然后敲击 `cin(` 。Vim会删除括号内的内容然后进入插入模式，并且光标会停留在括号的中间。

你可以将这个映射理解为“在下一个括号内(inside next parentheses)”。它会对当前行光标所在位置的下一个括号内的文本执行operator。

我们再创建一个“在上一个括号内(inside last parentheses)”的movement进行对照。（在这里使用“前一个(previous)“可能更准确，但这会覆盖“段落(paragraph)”movement）

```
:::vim
:onoremap il( :<c-u>normal! F)vi(<cr>
```

先试试确保这个命令可以工作。

那么这些映射是怎么工作的呢？首先， `<c-u>` 比较特殊，可以先不用管（你只需要相信我这个东西可以让这个映射在任何情况下都能正常工作）。如果我们删除它的话，这个映射会变成这个样子：

```
:::vim
:normal! F)vi(<cr>
```

`:normal!` 会在后面的章节谈到，现在你只需要知道它可以在常用模式下模拟按下按键。例如，运行 `:normal! dddd` 会删除两行，就像按下 `dddd` 。映射后面的 `<cr>` 是用来执行 `:normal!` 命令的。

那么现在我们可以认为这个映射的关键是运行下面这些按键组成的命令：

```
:::vim
F)vi(
```

This is fairly simple: 这个命令很容易理解：

- `F)` : 向后移动到最近的 `)` 字符。
- `vi(` : 进入可视模式选择括号内的所有内容。

这个movement结束在在可视模式下选择中我们想操作的文本，然后Vim会对选中的文本执行操作，就像通常情况一样。

一般规则

下面两条规则可以让你可以很直观的以多种方式创建operator-pending映射：

- 如果你的operator-pending映射以在可视模式下选中文本结束，Vim会操作这些文本。
- 否则，Vim会操作从光标的原始位置到一个新位置之间的文本。

练习

为"around next parentheses"和"around last parentheses"创建operator-pending映射

为打括号创建类似的in/around next/last的mappings。

阅读 `:help omap-info` ，看看你可不可以搞清楚 `<c-u>` 是干啥的。

更多Operator-Pending映射

Operators和movements所包含的理念是Vim中的一个非常重要的概念，也是Vim之所以这么高效的最大原因所在。在这一章我们会在这一块做更多的实践，这会让Vim变得更强大。

假设你现在在往Markdown中写入一些文字。如果你没有用过Markdown，不要紧，对于我们现在要做的事情而言，它很简单。把下面的文字输入到一个文件中：

```
:::markdown
Topic One
=========

This is some text about topic one.

It has multiple paragraphs.

Topic Two
=========

This is some text about topic two.  It has only one paragraph.
```

使用 `=` 作为“下划线”的行会被Markdown当作标题。我们现在创建一些映射，这些映射可以让我们通过movements定位到标题。运行下面的命令：

```
:::vim
:onoremap ih :<c-u>execute "normal! ?^==\\+$\r:nohlsearch\rkvg_"<cr>
```

这个映射有些复杂。现在把你的光标放到文本中的某个位置（不要放到标题上）,然后敲击 `cih` 。Vim会删除光标所在章节的标题，然后保持在插入模式，这可以称为"修改所在的标题(change inside heading)"。

这里使用了一些我们之前从来没有见过的东西，所以我们有必要单独分析下每一部分的含义。这个映射的第一部分， `:onoremap ih` 是映射命令，这个我们很熟悉了，无需多言。 `<c-u>` 上一章讲过，我们现在也不深究。

接着看看剩下的部分：

```
:::vim
:execute "normal! ?^==\\+$\r:nohlsearch\rkvg_"<cr>
```

Normal

`:normal` 命令的后面会跟着一串字符，无论这些字符表示什么含义， `:normal` 命令都会执行它们，就像是在常用模式下敲击这些字符一样。我们会在后面的章节中谈论关于 `:normal` 的更多细节，由于这个它已经出现多次，所以我觉得有必要现在做一个简单的说明，算是浅尝辄止吧。执行下面的命令：

```
:::vim
:normal gg
```

Vim会将光标跳转到文件的顶部。现在执行下面的命令：

```
:::vim
:normal >>
```

Vim将缩进当前行。

那 `normal` 后面的 `!` 是干啥的呢？先别管，以后再说。

Execute

`execute` 命令后面会跟着一个Vim脚本字符串（以后会深究它的细节），然后把这个字符串当作一个命令执行。执行下面的命令：

```
:::vim
:execute "write"
```

Vim会写文件，就像你已经输入了 `:write<cr>` 一样。现在执行下面的命令：

```
:::vim
:execute "normal! gg"
```

Vim会执行 `:normal! gg` ，这个会将光标跳转到文件的顶部，跟之前一样。现在问题来了，我们为什么要搞得这么蛋疼，又是 `execute` ，又是 `normal!` ，直接执行 `normal!` 不就可以搞定么？

看看下面的命令，猜猜它会干啥：

```
:::vim
:normal! gg/a<cr>
```

这个命令似乎会做下面的一些事情：

- 将光标跳转到文件的顶部。
- 准备搜索。
- 把“a”当作目标字符串进行搜索。
- 按下return键执行搜索。

你自己执行一下，Vim会将光标跳转到了文件顶部，然后。。没有然后了！

之所以会这样是由于 `normal!` 的一个问题，这问题是 `normal!` 不能识别“特殊字符”，例如这里的 `<cr>` 。这个问题有很多办法可以搞定，最简单的就是使用 `execute` ，另外使用 `execute` 也会让脚本更易读。

当 `execute` 碰到任何你想让它执行的字符串的时候。它会先替换这个字符串中的所有特殊字符。在这个示例中， `\r` 是一个转义字符，它表示的是"回车符（carriage return）"。两个反斜线也是一个转义字符，它会将一个反斜线当作一般字符放到字符串中。

如果我们按照上面的分析替换这个映射中的特殊字符，然后就可以搞清楚这个映射会怎么执行：

```
:::text
:normal! ?^==\+$<cr>:nohlsearch<cr>kvg_
                ^^^^           ^^^^
                 ||             ||
这里的<cr>实际上是一个回车符，而不是由4个字符——“左尖括号”，“c“，”r“和“右尖括号”组成的字符串。
```

所以现在 `normal!` 会执行这些字符，如同我们是在常用模式下敲击了它们一样。我们以回车符对这些字符进行拆分，然后看看它们是怎么执行的：

```
:::vim
?^==\+$
:nohlsearch
kvg_
```

第一部分 `?^==\+$` 会向后搜索任何由两个或多个等号组成的行，这些行不会包含除等号外的任何其他字符。这个命令执行后会让光标停留在符合搜索条件的行的行首。

之所以使用向后搜索，是因为当你想“修改所在的标题(change inside heading)”的时候，而当前光标是位于某一节的文本上，那么你最可能想做的是修改这一节的标题，而不是下一节的标题。

第二部分是 `:nohlsearch` 命令。这个命令只是简单的清除之前的搜索结果的高亮显示，防止这些高亮显示分散我们的注意。

最后一部分是三个常用模式下的命令的序列：

- `k` ：向上移动一行。第一部分已经将光标定位到了等号符号组成的行的第一个字符，所

以执行这个命令后光标就会被定位到标题的第一个字符上。
- `v` ：进入可视模式(characterwise)。
- `g_` ：移动到当前行的最后一个非空字符上。这里没有使用 `$` ，是因为 `$` 会选中换行符号，这不是我们所想要的。

结果

这个映射做了很多工作，所以看起来有些复杂，不过我们已经搞清楚了这个映射中的每个部分。现在来概括一下：

- 我们为“所在章节的标题内（inside this section's heading）”创建了一个operator-pending的映射。
- 我们使用了 `execute` 和 `normal!` 这两个命令来执行我们用于选择标题的常用命令，这让我们可以在这些命令中使用特殊字符。
- 我们的映射会搜索由等号组成的行从而定位到一个标题，然后在常用模式下选中标题的文本。
- Vim进行剩下的处理标题的工作。

再来看一个映射。执行下面的命令：

```
:::vim
:onoremap ah :<c-u>execute "normal! ?^==\\+\r:nohlsearch\rg_vk0"<cr>
```

把光标放到某一节的文字上，然后敲击 `cah` 试试看。这一次Vim不仅会删除这一节的标题，而且还会删除跟这个标题相连的等号组成的行。你可以把这个movement当作是“环绕这一节的标题(*around* this section's heading)“。

这个映射有什么不同呢？让我们对照之前的映射看一下：

```
:::vim
:onoremap ih :<c-u>execute "normal! ?^==\\+$\r:nohlsearch\rkvg_"<cr>
:onoremap ah :<c-u>execute "normal! ?^==\\+$\r:nohlsearch\rg_vk0"<cr>
```

唯一的不同是映射的后面用于选择文本的部分：

```
:::text
inside heading: kvg_
around heading: g_vk0
```

其他的部分都是一模一样的，所以我们现在从将光标定位到等号组成的行的第一个字符的那个部分开始进行讲解：

- `g_` ：移动到当前行（译注：等号组成的行）的最后一个非空字符。
- `v` ：进入可视模式(characterwise)。
- `k` ：向上移动一行。这会将光标移动到包含标题文字的行上。
- `0` ：移动到这一行（译注：标题行）的第一个字符。

这一系列命令的执行结果就是在可视模式下同时选中标题的文字和等号组成的行，然后Vim会在这两行上执行相应的操作。

练习

Markdown也可以用 `-` 字符来限定标题。调整上面的正则表达式使得这些映射可以工作在不同类型的标题上。你可能想查看 `:help pattern-overview` 。记住正则表达是在一个字符串中，所以反斜线需要进行转义。

添加两个创建这些映射的自动命令到你的 `~/.vimrc` 文件中。确保只对合适的缓冲区使用这些映射，并且确保使用自动命令组来防止每次加载 `~/.vimrc` 的时候创建这些自动命令的副本。

阅读 `:help normal` 。

阅读 `:help execute` 。

阅读 `:help expr-quote` 了解你可以在字符串中使用的转义序列。

创建一个“在下一个邮件地址内(inside next email address)”的operator-pending映射，然后你就可以使用“修改在下一个邮件地址内(change inside next email address)”。将 `in@` 作为映射的按键是个不错的选择。你可能还需要将这个按键映射为 `/...some regex...<cr>` 。

状态条

Vim允许自定义每个窗口底部的状态条显示的文字，你可以通过设置 `statusline` 选项来进行自定义。执行下面的命令：

```
:::vim
:set statusline=%f
```

你可以在状态条上看到当前所编辑文件的路径（相对于当前路径）。再执行这个命令：

```
:::vim
:set statusline=%f\ -\ FileType:\ %y
```

现在你可以在状态条中看到类似 `foo.markdown - FileType: [markdown]` 这样的文字。

如果你熟悉C语言中的 `printf` 或者Python的字符串插值，那么这个选项的格式看起来会比较眼熟。如果不熟悉，你只需要记住以 `%` 开头的字符串会被展开为不同的文字，这取决于 `%` 后面的字符。在上面的示例中， `%f` 会被替换为文件名， `%y` 会被替换为文件类型。

注意状态条中的空格需要反斜线进行转义，这是因为 `set` 可以同时设置多个选项，这些选项会用空格分隔，我们在第二章讲过这个。

状态条设置可以很快变得非常复杂，不过有一个更好的办法来设置它们以至于让它们看起来更清晰。执行下面的命令：

```
:::vim
:set statusline=%f         " 文件的路径
:set statusline+=\ -\      " 分隔符
:set statusline+=FileType: " 标签
:set statusline+=%y        " 文件的类型
```

第一个命令使用 `=` 来设置状态条只显示文件名，从而将之前的所有会在状态条中显示的值都屏蔽掉。之后再使用 `+=` 逐渐添加其他要显示的内容，一次添加一条。同时还使用注释来说明每一条所表示的含义以方便其他的人阅读我们的代码（也会方便我们自己以后阅读）。

执行下面的命令：

```
:::vim
:set statusline=%l    " 当前行号
:set statusline+=/    " 分隔符
:set statusline+=%L   " 总行数
```

现在状态条只包含当前所在行以及文件的总行数，并且显示成 `12/223` 这个样子。

宽度和边距

可以在 `%` 后面添加额外的字符来改变状态条中信息的显示样式。执行下面的命令：

```
:::vim
:set statusline=[%4l]
```

现在状态条中的文件行数会至少显示为4个字符的宽度（例如：`[  12]`），这可以用于防止状态条中的文字总是令人厌烦地跳来跳去。

默认情况下在值的左边添加边距。执行下面的命令：

```
:::vim
:set statusline=Current:\ %4l\ Total:\ %4L
```

你的状态条看来会是这个样子：

```
:::text
Current:   12 Total:  223
```

你可以使用 `-` 将边距添加在右边，而不是左边。执行下面的命令：

```
:::vim
:set statusline=Current:\ %-4l\ Total:\ %-4L
```

现在你的状态条看起来会是这个样子：

```
:::text
Current: 12   Total: 223
```

这样就好看多了，因为数字值是紧挨着它的标签的。

对于会被显示为数字的代码，你可以让Vim使用0代替空格来填充边距。执行下面的命令：

```
:::vim
:set statusline=%04l
```

现在当光标位于第12行的时候你的状态条会显示 `0012` 。

最后，你可以设置一个代码所要输出的值的最大宽度。执行下面命令：

```
:::vim
:set statusline=%F
```

`%F` 会显示当前文件的完整路径。现在执行下面的命令改变最大宽度：

```
:::vim
:set statusline=%.20F
```

如果有必要路径会被删简，像下面这样：

```
:::text
<hapters/17.markdown
```

这可以用于防止路径或者是其他的很长的代码占用整个行。

通用格式

阅读 `:help statusline` 查看状态条中代码的通用格式：

```
:::text
%-0{minwid}.{maxwid}{item}
```

除了 `%` 和item外其他都是可选的。

分割

我们不会探讨状态条的更多细节（Vim的文档有非常详细的说明，如果你想学到更多，建议阅读它们），不过我们会介绍一个简单的代码，这个代码可以立即带来价值。执行下面的命令：

```
:::vim
:set statusline=%f         " 文件的路径
:set statusline+=%=        " 切换到右边
:set statusline+=%l        " 当前行
:set statusline+=/         " 分隔符
:set statusline+=%L        " 总行数
```

现在状态条的左边会包含文件的路径，当前行/总行数会显示在状态条的右边。 `%=` 这个代码告诉Vim所有在此之后要在状态条上显示的信息都应该右对齐（作为一个整体），而不是左对齐。

练习

浏览 `:help statusline` 中的可用代码。先别在意那些你现在理解不了的代码。

编辑你的 `~/.vimrc` 文件创建一个自定义的状态条。确保在 `set` 中使用 `+=` 来一条一条地定义要显示的代码，并且每一行的设置添加注释来说明每一条的含义。

尝试使用自动命令和 `setlocal` 来为不同的文件定义不同的状态条。确保使用了自动命令组防止自动命令被重复创建（永远记住）。

负责任的编码

到目前为止我们已经介绍了一堆Vim命令，这可以让你可以快速自定义Vim。除了自动命令组外其他的命令都是单行的命令，你可以不费吹灰之力就把它们添加到你的 `~/.vimrc` 文件中。

这本书的下一部分我们会开始专注于Vim脚本编程，将其当作一个真正的编程语言对待，不过在此之前，我会先讲一些在编写大量的Vim脚本时需要注意的东西。

注释

Vim脚本非常强大，但对于那些想进入这个领域的程序员而言，在最近几年它似乎逐渐变得像一个弯弯曲曲的迷宫，让进入的人找不到归路。

Vim的选项和命令经常会比较简短生硬，并且难于阅读，另外处理兼容性问题也会增加代码的复杂度。编写一个插件并且允许用户自定义又会让复杂度更进一级。

在编写大量Vim脚本时要保持防御意识。要养成习惯添加注释说明某段代码是干什么的，如果有一个相关的帮助主题（help topic），最好在注释中说明！

这不仅会给你以后的维护带来方便，而且如果你将你的 `~/.vimrc` 文件分享到Bitbucket或者GitHub（强烈推荐你这么做），这些注释也会帮助其他的人理解你的脚本。

分组

之前创建的映射可以让我们在使用Vim的同时方便快捷地编辑和加载 `~/.vimrc` 。不幸的是这会导致 `~/.vimrc` 中的代码快速增长以至失去控制，并且变得难于阅读浏览。

我们用于对付这种情况的方法是使用Vim的代码折叠功能，将多行代码组织起来的作为一个部分然后对这部分的代码进行折叠。如果你从来没有用过Vim的折叠功能，那么你现在应该尽快去瞄一瞄。很多人（包括我自己）都认为在日常编码工作中代码折叠是不可或缺的。

首先我们需要为Vim脚本文件设置折叠。在你的 `~/.vimrc` 文件中添加下面几行：

```
:::vim
augroup filetype_vim
    autocmd!
    autocmd FileType vim setlocal foldmethod=marker
augroup END
```

这会告诉Vim对任何Vim脚本文件使用 `marker` 折叠方法。

现在在显示 `~/.vimrc` 文件的窗口中执行 `:setlocal foldmethod=marker` 。如果你不执行这个命令，你会发现加载 `~/.vimrc` 文件后没什么效果，这是因为Vim已经为这个文件设置了文件类型(FileType)，而自动命令只会在设置文件类型的时候执行。这让你以后不需要手动来做这个事情。

现在在自动命令组开始和结束的地方添加两行，像下面这样：

```
:::vim
" Vimscript file settings ---------------------- {{{
augroup filetype_vim
    autocmd!
    autocmd FileType vim setlocal foldmethod=marker
augroup END
" }}}
```

切换到常用模式，将光标放到这些文字中的任意一行，然后敲击 `za` 。Vim会折叠从包含 `{{{` 的行到包含 `}}}` 的行之间的所有行。再敲击 `za` 会展开所有这些行。

刚开始你可能会觉得为了代码折叠而对源代码进行注释会有些不合理，我刚开始也这么想。对于大多数文件我现在仍然觉得这种做法并并不合适。因为不是所有人都使用相同的编辑器，所以在代码中添加的折叠注释对于那些不用Vim的人而言就像是噪音。

不过Vim脚本文件比较特殊，因为一个不用Vim的人不太可能会读你的代码，并且最重要的是如果不对代码进行分组处理，写着写着你就不知道写到哪里了，严重点可能会经脉尽断，吐血而亡。

先自己尝试尝试吧，说不定你会逐渐喜欢上它。

简短的名称(Short Names)

对于大多数命令和选项，Vim支持使用它们的缩写。例如，下面的两个命令做的事情完全一样：

```
:::vim
:setlocal wrap
:setl wrap
```

我强烈提醒你不要在你的 `~/.vimrc` 或者是你编写的插件中使用这些缩写。Vim脚本对于初学者而言本来就已经够晦涩难懂了；从长远来看使用缩写只会使得它更难于阅读。即使你知道某个缩写的意思，其他人未必读得懂。

换句话说，缩写只在编码的过程中手动执行命令的时候会很有用。在你按了回车键以后，就没人会看到你输入什么了，这样你也没必要输入更多的字符。

练习

检查你的 `~/.vimrc` 文件，将所有相关的行组织起来。你可以这么开头："基本设置(Basic Settings)"，"文件类型相关设置(FileType-specific settings)"，"映射(Mappings)"，和"状态条(Status Line)"。然后在每个部分添加折叠标记和标题。

想想怎么让Vim在第一次打开 `~/.vimrc` 文件的时候自动折叠所有设置了折叠注释的行。阅读 `:help foldlevelstart` 你会知道怎么搞。

检查你的 `~/.vimrc` 文件，把所有的命令和选项的缩写改成全称。

检查你的 `~/.vimrc` 文件，确保里面没有什么敏感信息。然后创建一个git或者Mercurial仓库，再将 `~/.vimrc` 文件放到里面，然后将这个文件链接到 `~/.vimrc` 。

提交你刚才创建的仓库，并把它放到Bitbucket或者GitHub上，这样其他的人都可以看到和学习它。记住要经常提交和推送到仓库中，这样你所做的修改也会被记录下来。

如果你不只在一个机器上使用Vim，那你就可以克隆那个仓库，然后像之前一样将这个文件链接到 `~/.vimrc` 文件。这样你就可以在所有的机器上都使用同样的Vim配置了。

变量

到目前为止我们已经讲完了单行命令。在本书后面的三分之一个章节中将会把Vim脚本当作一个脚本语言。这部分东西不会像前面的你学到的东西一样马上可以学以致用，不过这是为本书的最后一部分打基础，最后一部分会讲解创建一个插件所需要的各个方面的东西。

我们开始吧。我们首先要了解的是变量。执行下面的命令。

```
:::vim
:let foo = "bar"
:echo foo
```

Vim会显示 `bar` 。 `foo` 现在是一个变量，我们将一个字符串 `"bar"` 赋值给它。现在执行这些命令：

```
:::vim
:let foo = 42
:echo foo
```

Vim会显示 `42` ，因为我们将 `foo` 赋值为整型 `42` 。

从这些小例子似乎可以看出Vim脚本是动态类型的。事实并非如此，我们之后会说明。

作为变量的选项

你可以通过一种特殊语法将选项作为变量来设置。执行下面的命令：

```
:::vim
:set textwidth=80
:echo &textwidth
```

Vim会显示 `80` 。在名称的前面加一个 `&` 符号是告诉Vim你正在引用这个选项，而不是在使用一个名称刚好相同的变量。

我们来看下Vim是怎么处理布尔选项的。执行下面的命令：

```
:::vim
:set nowrap
:echo &wrap
```

Vim显示 `0` 。然后再试试这些选项：

```
:::vim
:set wrap
:echo &wrap
```

这次Vim会显示 `1` 。这些输出很明确提示Vim会将整型 `0` 当作"false"，整型 `1` 当作"true"。我们可以更进一步假设Vim会将所有的非0值整型当作"truthy"，而事实确实如此。

我们也可以使用 `let` 命令来设置作为变量的选项。执行下面的命令：

```
:::vim
:let &textwidth = 100
:set textwidth?
```

Vim会显示 `textwidth=100` 。

既然 `set` 可以搞定选项的设置，那我们为什么还要用 `let` 呢？执行下面的命令：

```
:::vim
:let &textwidth = &textwidth + 10
:set textwidth?
```

这一次Vim显示 `textwidth=110` 。当你用 `set` 来设置某个选项，你只能给它设置一个常量值。当你使用 `let` 并将它作为一个变量来设置，你可以使用Vim脚本的所有强大之处来决定它的值。

本地选项

如果你想将某个选项作为变量来设置它的本地值，而不是全局值，你需要在变量名前面加前缀。

在两个分隔的窗口中分别打开两个文件。执行下面的命令：

```
:::vim
:let &l:number = 1
```

然后切换到另一文件，然后再执行下面的命令：

```
:::vim
:let &l:number = 0
```

注意第一个窗口会出现行号，而第二个没有。

作为变量的寄存器(Register)

你也可以将寄存器当作变量来读取和设置。执行下面的命令：

```
:::vim
:let @a = "hello!"
```

现在把光标放到文本中的某个地方然后敲击 `"ap` 。这个命令会告诉Vim“在这里粘贴寄存器 `a` 中的内容”。我们设置了这个寄存器的内容，所以Vim会将 `hello!` 粘贴到你的文本中。

还可以读寄存器的内容。执行下面的命令：

```
:::vim
:echo @a
```

Vim会输出 `hello!` 。

在你的文件中选择一个单词然后用 `y` 复制，再执行下面的命令：

```
:::vim
:echo @"
```

Vim会输出你刚才复制的单词。 `"` 寄存器是“未命名(unnamed)”寄存器，在复制的时候没有指定寄存器的文本都会放到这里。

在你的文件中执行搜索 `/someword` ，然后执行下面的命令：

```
:::vim
:echo @/
```

Vim会输出你刚刚使用的搜索模式。这样你就可以通过编程来读和修改当前的搜索模式，有些时候这会很有用。

练习

检查你的 `~/.vimrc` 文件，然后将其中的一些 `set` 和 `setlocal` 命令替换为它们的 `let` 形式。记住布尔选项仍然需要被设置为某个值。

尝试将某个布尔选项设置为0和1之外的值，例如 `wrap` 。当你将它设置为一个不同的数字时会怎么样？如果设置为字符串又会是什么情况？

回到你的 `~/.vimrc` 文件，然后恢复所有的修改。在 `set` 可以搞定的时候，永远都不要用 `let` ，这是因为 `let` 更难于阅读。

阅读 `:help registers` ，然后看看你可以进行读和写的寄存器列表。

变量作用域

如果你之前用过像Python或者Ruby之类的动态语言，现在你可能已经熟悉了Vim脚本的变量。你会发现Vim变量的大部分内容跟你想的一样，不过有一个东西可能会不同，那就是变量的作用域。

在两个分隔的窗口中分别打开两个不同的文件，然后在其中一个窗口中执行下面的命令：

```
:::vim
:let b:hello = "world"
:echo b:hello
```

如你所愿，Vim会显示 `world` 。现在切换到另外一个缓冲区再次执行 `echo` 命令：

```
:::vim
:echo b:hello
```

这一次Vim会抛出一个无法找到变量的错误，

当你在变量名中使用 `b:` ，这相当于告诉Vim变量 `hello` 是当前缓冲区的本地变量。

Vim有很多不同的变量作用域，不过在使用其他类型变量作用域之前我们需要先学习更多Vim脚本编程的知识。就目前而言，你只需要记住当某个变量由一个字符和冒号开头，那么这就表示它是一个作用域变量。

练习

浏览 `:help internal-variables` 中的作用域列表。先看看，熟悉熟悉，即使有不明白的地方也没关系。

条件语句

每种编程语言都有产生分支流程的方法，在Vimscript中，这是用 `if` 语句实现的。 `if` 语句是Vimscript中产生分支的基本方法。这里没有类似Ruby中的 `unless` 语句， 所以代码中所有的判断都需要用 `if` 实现。

在谈论Vim的 `if` 语句之前，我们需要花费额外的时间讲讲语法，这样可以在同一页里讲完它。

多行语句

有时你在一行里写不下所需的Vimscript。在讲到自动命令组时，我们已经遇到过这样的例子了。 这里是我们之前写过的代码：

```
:::vim
:augroup testgroup
:    autocmd BufWrite * :echom "Baz"
:augroup END
```

在理想的情况下，你可以分开成三行来写。但在手工执行命令的时候，这样写就太冗长了。其实，你可以用管道符(`|`)来隔开每一行。执行下面的命令：

```
:::vim
:echom "foo" | echom "bar"
```

Vim会把它当作两个独立的命令。如果你看不到两行输出，执行 `:messages` 查看消息日志。

在本书的剩余部分，当你想手工执行一个命令，却对输入新行和冒号感到心烦时，试试用管道隔开， 在一行里写完。

If的基本用法

现在让我们回到正题上来，执行下面的命令：

```
:::vim
:if 1
:    echom "ONE"
:endif
```

Vim将显示`ONE`，因为整数`1`是"truthy"。现在执行下面命令：

```
:::vim
:if 0
:    echom "ZERO"
:endif
```

Vim将不显示`ZERO`，因为整数`0`是"falsy"。让我们看看对字符串是怎么处理的。执行下面命令：

```
:::vim
:if "something"
:    echom "INDEED"
:endif
```

结果可能让你吃惊。Vim不会把非空字符串当作"truthy"，所以什么也没有显示。

让我们打破沙锅问到底。执行下面的命令：

```
:::vim
:if "9024"
:    echom "WHAT?!"
:endif
```

这次Vim会显示了！为什么会这样？

为了搞懂发生了什么，执行下面三个命令：

```
:::vim
:echom "hello" + 10
:echom "10hello" + 10
:echom "hello10" + 10
```

第一个命令使得Vim输出`10`，第二个命令输出`20`，第三个则又一次输出`10`！

在探究了所有的命令后，对于Vimscript我们可以得出结论：

- 如有必要，Vim将强制转换变量(和字面量)的类型。在解析`10 + "20foo"`时，Vim将把`"20foo"`转换成一个整数(`20`)然后加到`10`上去。
- 以一个数字开头的字符串会被强制转换成数字，否则会转换成`0`
- 在所有的强制转换完成后，当`if`的判断条件等于非零整数时，Vim会执行`if`语句体。

Else 和 Elseif

Vim，像Python一样，支持"else"和"else if"分句。执行下面的命令：

```
:::vim
:if 0
:    echom "if"
:elseif "nope!"
:    echom "elseif"
:else
:    echom "finally!"
:endif
```

Vim输出`finally!`，因为前面的判断条件都等于0，而0代表falsy。

练习

来一杯啤酒，安抚自己因Vim中的字符串强制转换而受伤的心。

比较

我们已经学习了条件语句，但如果我们不能进行比较， `if` 语句并不怎么有用。 当然Vim允许我们比较值的大小，只是不会像看上去那么一目了然。

执行下面的命令：

```
:::vim
:if 10 > 1
:    echom "foo"
:endif
```

显然，Vim会显示 `foo` 。现在执行下面的命令：

```
:::vim
:if 10 > 2001
:    echom "bar"
:endif
```

Vim什么都不显示，因为 `10` 不比 `2001` 大。目前为止，一切正常。运行下面命令：

```
:::vim
:if 10 == 11
:    echom "first"
:elseif 10 == 10
:    echom "second"
:endif
```

Vim显示 `second` 。没什么好惊讶的。让我们试试比较字符串。执行下面命令：

```
:::vim
:if "foo" == "bar"
:    echom "one"
:elseif "foo" == "foo"
:    echom "two"
:endif
```

Vim输出 `two` 。还是没什么好惊讶的，所以我开头说的(译注：Vim的比较不像看上去那么直白)到底是指什么呢？

大小写敏感

执行下面的命令：

```
:::vim
:set noignorecase
:if "foo" == "FOO"
:    echom "vim is case insensitive"
:elseif "foo" == "foo"
:    echom "vim is case sensitive"
:endif
```

Vim执行 `elseif` 分句,所以显然Vimscript是大小写敏感的。有道理，但没什么好震惊的。 现在执行下面命令：

```
:::vim
:set ignorecase
:if "foo" == "FOO"
:    echom "no, it couldn't be"
:elseif "foo" == "foo"
:    echom "this must be the one"
:endif
```

啊！ 就在这里停下来。是的，你所见属实。

`==` 的行为取决于用户的设置。

我发誓我没忽悠你。你再试试看看。我没开玩笑，这不是我干的。

防御性编程

这意味着什么？意味着在为别人开发插件时，你不能信任 `==` 。 一个不加包装的 `==` 不能出现在你的插件代码里。

这个建议就像是" `nmap` VS `nnoremap` "一样。永远不要猜测你的用户的配置。 Vim既古老，又博大艰深。在写插件时，你不得不假定用户们的配置五花八门，千变万化。

所以怎样才能适应这荒谬的现实？好在Vim有额外两种比较操作符来处理这个问题。

执行下面的命令：

```
:::vim
:set noignorecase
:if "foo" ==? "FOO"
:    echom "first"
:elseif "foo" ==? "foo"
:    echom "second"
:endif
```

Vim显示 `first` 因为 `==?` 是"无论你怎么设都大小写不敏感"比较操作符。现在执行下面的命令：

```
:::vim
:set ignorecase
:if "foo" ==# "FOO"
:    echom "one"
:elseif "foo" ==# "foo"
:    echom "two"
:endif
```

Vim显示 `two` 因为 `==#` 是"无论你怎么设都大小写敏感"比较操作符。

故事的最后告诉我们一个道理：你应该总是用显式的大小写敏感或不敏感比较。 使用常规的形式是错的并且它终究会出错。打多一下就能拯救你自己于焦头烂额中。

当你比较整数时，这点小不同不会有什么影响。 不过，我还是建议每一次都使用大小写敏感的比较(即使不一定需要这么做)，好过该用的时候忘记用了。

在比较整数时使用 `==#` 或 `==?` 都可以，而且将来一旦你改成字符串间的比较，它还会正确工作。 如果你真想用 `==` 比较整数也不是不行，不过要铭记，一旦被改成字符串间的比较,你需要修改比较操作符。

练习

尝试 `:set ignorecase` 和 `:set noignorecase` ，看看在不同状态下比较的表现。

阅读 `:help ignorecase` 来看看为什么有的人设置了这个选项。

阅读 `:help expr4` 看看所有允许的比较操作符。

函数

一如大多数编程语言，Vimscript支持函数。让我们看看如何创建函数，然后再讨论它们的古怪之处。

执行下面的命令：

```
:::vim
:function meow()
```

你可能会认为这将定义函数 `meow` 。不幸的是，情况不是这样的，我们已经掉进了Vimscript其中的一个坑。

没有作用域限制的**Vimscript**函数必须以一个大写字母开头！

即使你真的给函数限定了作用域(我们待会会谈到)，你最好也用一个大写字母开头。 大多数Vimscript程序猿都是这么做的，所以不要破例。

ok，是时候认真地定义一个函数了。执行下面的命令：

```
:::vim
:function Meow()
:  echom "Meow!"
:endfunction
```

这次Vim愉快地定义了一个函数。让我们试试运行它：

```
:::vim
:call Meow()
```

不出所料，Vim显示 `Meow!`

让我们试试令它返回一个值。执行下面的命令：

```
:::vim
:function GetMeow()
:  return "Meow String!"
:endfunction
```

现在执行这个命令试试：

```
:::vim
:echom GetMeow()
```

Vim将调用这个函数并把结果传递给 `echom` ，显示 `Meow String!` 。

调用函数

我们已经看到，Vimscript里调用函数有两种不同的方法。

当你想直接调用一个函数时，使用 `call` 命令。执行下面命令：

```
:::vim
:call Meow()
:call GetMeow()
```

第一个函数输出 `Meow!` ，然而第二个却没有任何输出。当你使用 `call` 时，返回值会被丢弃，所以这种方法仅在函数具有副作用时才有用。

第二种方法是在表达式里调用函数。这次不需要使用 `call` ，你只需引用函数的名字。 执行下面的命令：

```
:::vim
:echom GetMeow()
```

正如我们见过的，这会调用 `GetMeow` 并把返回值传递给 `echom` 。

隐式返回

执行下面的命令：

```
:::vim
:echom Meow()
```

这将会显示两行： `Meow!` 和 `0` 。第一个显然来自于 `Meow` 内部的 `echom` 。第二个则告诉我们， 如果一个Vimscript函数不返回一个值，它隐式返回 `0` 。看我们可以利用这一点做什么。执行下面命令：

```
:::vim
:function TextwidthIsTooWide()
:  if &l:textwidth ># 80
:    return 1
:  endif
:endfunction
```

这个函数涉及到我们之前学到的许多重要概念：

- `if` 语句
- 将选项作为变量
- 访问特定作用域里的选项变量
- 大小写敏感的比较

如果你对以上内容感到陌生，最好翻到前几章温习一遍。

现在我们已经定义了一个函数，该函数告诉我们当前缓冲区的 `textwidth` 会不会设得'太过宽'。 (因为80字符的限制适用于除了HTML之外的任何代码文件)

现在让我们使用它。执行下面的命令：

```
:::vim
:set textwidth=80
:if TextwidthIsTooWide()
:  echom "WARNING: Wide text!"
:endif
```

在这里我们做了什么？

- 一开始我们设置全局的 `textwidth` 为 `80` 。
- 接着我们运行一个if语句判断 `TextwidthIsTooWide()` 是否为真。
- 由于不满足条件， `if` 语句体(译注：包括函数内的和函数外的)不会被执行。

因为我们没有显式返回一个值，Vim从函数中返回代表'falsy'的 `0` 。试试改变一下。运行下面的命令：

```
:::vim
:setlocal textwidth=100
:if TextwidthIsTooWide()
:  echom "WARNING: Wide text!"
:endif
```

这次函数中的 `if` 执行了它的语句体，返回 `1` ，并且我们手工输入的 `if` 语句也执行了它的语句体。

练习

阅读 `:help :call` 。目前先忽略关于"范围"的内容。你可以传递多少参数给一个函数？感到惊讶不？

阅读 `:help E124` 第一自然段并找出你可以用哪些字符来命名函数。可以用下划线吗？点(Dashes)呢？ 重音符号(Accented characters)？Unicode符号？如果读了文档还是搞不清楚，试一下看看。

阅读 `:help return` 。这个命令的缩写("short form")是什么？(我说了你千万不要用它) 在你的预期之内吗？如果不是，为什么？

函数参数

毫无疑问，Vimscript函数可以接受参数。执行下面的命令：

```
:::vim
:function DisplayName(name)
:  echom "Hello!  My name is:"
:  echom a:name
:endfunction
```

执行下面的函数：

```
:::vim
:call DisplayName("Your Name")
```

Vim将显示两行：`Hello! My name is:` 和 `Your Name`。

注意我们传递给`echom`命令的参数前面的`a:`。这表示一个变量的作用域，在前几章(译注：第20章)我们曾讲过。

让我们试一下不带作用域前缀会怎么样。执行下面的命令：

```
:::vim
:function UnscopedDisplayName(name)
:  echom "Hello!  My name is:"
:  echom name
:endfunction
:call UnscopedDisplayName("Your Name")
```

这次Vim抱怨说它找不到变量`name`。

在写需要参数的Vimscript函数的时候，你总需要给参数加上前缀`a:`，来告诉Vim去参数作用域查找。

可变参数

Vimscript函数可以设计为接受不定数目的参数，就像Javascript和Python中的一样。执行下面命令：

```
:::vim
:function Varg(...)
:  echom a:0
:  echom a:1
:  echo a:000
:endfunction

:call Varg("a", "b")
```

这个函数向我们展示了许多东西，让我们来逐一审视。

函数定义中的 `...` 说明这个函数可以接受任意数目的参数。就像Python函数中的 `*args`

函数中的第一行为输出消息 `a:0` ，结果显示 `2` 。当你在Vim中定义了一个接受可变参数的函数， `a:0` 将被设置为你额外给的参数数量(译注：注意是额外的参数数量)。 刚才我们传递了两个参数给 `Varg` ，所以Vim显示 `2` 。(译注：2 - 0 ==# 2)

第二行为输出 `a:1` ，结果显示 `a` 。你可以使用 `a:1` , `a:2` 等等来引用你的函数接受的每一个额外参数。 如果我们用的是 `a:2` ，Vim就会显示"b"

第三行有些费解。当一个函数可以接受可变参数， `a:000` 将被设置为一个包括所有传递过来的额外参数的列表(list)。 我们还没有讲过列表，所以不要太纠结于此。你不能对列表使用 `echom` ，因而在这里用 `echo` 代替。

你也可以将可变参数和普通参数一起用。执行下面的命令：

```
:::vim
:function Varg2(foo, ...)
:  echom a:foo
:  echom a:0
:  echom a:1
:  echo a:000
:endfunction

:call Varg2("a", "b", "c")
```

我们可以看到Vim将 `"a"` 作为具名参数(named argument) `a:foo` 的值，将余下的塞进可变参数列表中。

赋值

试试执行下面的命令：

```
:::vim
:function Assign(foo)
:  let a:foo = "Nope"
:  echom a:foo
:endfunction

:call Assign("test")
```

Vim将抛出一个错误，因为你不能对参数变量重新赋值。现在执行下面的命令：

```
:::vim
:function AssignGood(foo)
:  let foo_tmp = a:foo
:  let foo_tmp = "Yep"
:  echom foo_tmp
:endfunction

:call AssignGood("test")
```

这次就可以了，Vim显示 `Yep` 。

练习

阅读 `:help function-argument` 的前两段。

阅读 `:help local-variables` 。

数字

现在是时候开始深入讨论你能用到的变量类型。首先从数值类型开始吧。

Vimscript有两种数值类型：Number和Float。一个Number是32位带符号整数。一个Float是浮点数。

数字(Number)形式

你可以通过一些不同的方式设置Number的格式。执行下面的命令：

```
:::vim
:echom 100
```

没什么好惊讶的 -- Vim显示 `100` 。现在执行下面的命令：

```
:::vim
:echom 0xff
```

这次Vim显示 `255` 。你可以加 `0x` 或 `0X` 前缀来指定16进制的数字。现在执行下面的命令：

```
:::vim
:echom 010
```

你也可以加 `0` 前缀来使用八进制。不过由于容易混淆，用的时候要保持头脑清醒。尝试执行下面的命令：

```
:::vim
:echom 017
:echom 019
```

第一个命令中，Vim将打印出 `15` ，因为 `17` 在八进制中等于十进制的 `15` 。 在第二个命令中，Vim把数字的进制当作十进制，即使它以 `0` 开头，因为它不可能是一个八进制数字。

因为Vim会一声不吭地处理掉这样的错误，我建议尽量避免使用八进制数字。

浮点数(Float)格式

Float也可以用许多方式进行定制。执行下面的命令：

```
:::vim
:echo 100.1
```

注意这里我们使用了 `echo` 而不是更常用的 `echom` 。待会我会解释为什么这样做(译注：当然你现在可以试试看)。

Vim如愿输出了 `100.1` 。你也可以指定指数形式。执行下面命令：

```
:::vim
:echo 5.45e+3
```

Vim输出 `5450.0` 。也可以用负的指数。执行下面命令：

```
:::vim
:echo 15.45e-2
```

Vim输出 `0.1545` 。在10的幂前面的 `+` 或 `-` 是可选的。如果没有，就默认为正数。执行下面的命令：

```
:::vim
:echo 15.3e9
```

Vim将输出等价的 `1.53e10` 。小数点和小数点后面的数字是必须要有的。执行下面命令并看它为何出错：

```
:::vim
:echo 5e10
```

强制转换

当你在运算，比较或其他操作中混合使用Number和Float类型，Vim将把Number转换成Float， 以Float格式作为结果。执行下面命令：

```
:::vim
:echo 2 * 2.0
```

Vim输出 `4.0` 。

除法

在两个Number之间的除法中，余数会被丢弃。执行下面命令：

```
:::vim
:echo 3 / 2
```

Vim输出 `1` 。如果你希望Vim使用浮点数除法，至少有一个数字必须是Float， 这样剩下的数字也会被转换成浮点数。执行下面命令：

```
:::vim
:echo 3 / 2.0
```

Vim输出 `1.5` 。 `3` 被强制转换成一个浮点数，然后运行了普通的浮点数除法。

练习

阅读 `:help Float` 。什么情况下在Vimscript中不能用浮点数？

阅读 `:help floating-point-precision` 。这意味着你在写一个处理浮点数的Vim插件时需要注意什么？

字符串

接下来我们讲字符串类型。鉴于Vim是用来编辑文本的，你将频繁地跟这一类型打交道。

执行下面的代码：

```
:::vim
:echom "Hello"
```

Vim将输出 `Hello` 。目前为止，依然如此。

连接(Concatenation)

在日常编程中你经常需要把字符串连接到一起。执行下面的命令：

```
:::vim
:echom "Hello, " + "world"
```

发生什么了？不知为何，Vim显示 `0` ！

这就是问题所在：Vim的 `+` 运算符仅仅适用于数值。当你把一个字符串作为 `+` 的参数时，Vim会在执行加法前尝试把它强制转换成一个Number。(译注：在21章的练习里你真的喝了啤酒没) 执行下面的命令：

```
:::vim
:echom "3 mice" + "2 cats"
```

这次Vim显示 `5` ，因为字符串被相应地转换成 `3` 和 `2` 。

当提到"Number"时，我明确指的是Number类型。Vim不会强制转换字符串为Float类型！ 为求证我的话，试试下面的命令：

```
:::vim
:echom 10 + "10.10"
```

Vim显示 `20` ，因为在强制转换 `10.10` 成一个Number时，小数点后的内容都被丢弃了。

你需要使用连结运算符来连接字符串。执行下面命令：

```
:::vim
:echom "Hello, " . "world"
```

这次Vim显示 `Hello, world` 。 `.` 是Vim中的"连接字符串"运算符，可以用来连接字符串。 它不会在其间插入空格或别的什么东西。

Vim会在 `.` 两边进行强制转换。试试这个命令：

```
:::vim
:echom 10 . "foo"
```

Vim将显示 `10foo` 。首先它把 `10` 强制转换成字符串，接着把它跟右边的字符串连接在一起。不过当涉及到Float类型时，情况有点糟糕。执行这个命令：

```
:::vim
:echom 10.1 . "foo"
```

这次Vim抛出一个错误，声称我们把Float当作String了。Vim乐于让你在执行加法时把String当作Float， 却不爽你在连接字符串时把Float当作String。

故事的最后告诉我们一个道理：Vim酷似Javascript：它有时允许你逍遥自在地马虎对待类型差异， 但切莫这样做：因为出来混，迟早都要还的。(because it will come back to bite you at some point)

写Vimscript的时候，确信你清楚写下的每一个变量的类型。如果需要改变变量类型，你就得使用一个函数显式改变它， 即使那不是必要的。不要依赖Vim的强制转换，毕竟世上没有后悔药。

特殊字符

就像大多数编程语言，Vimscript允许你在字符串中使用转义字符串来表示"打不了"的字符。执行下面命令：

```
:::vim
:echom "foo \"bar\""
```

字符串中的 `\"` 将如你所愿地被替换成双引号。转义字符串在大多数情况下都会如你所愿。执行下面的命令：

```
:::vim
:echom "foo\\bar"
```

Vim显示`foo\bar`，因为`\\`是表示'\'的转义字符串，一如大多数编程语言。 现在执行下面的命令(注意那是`echo`而不是`echom`)：

```
:::vim
:echo "foo\nbar"
```

这次Vim将显示两行，`foo`和`bar`，因为`\n`会被替换为换行。现在试试下面的命令：

```
:::vim
:echom "foo\nbar"
```

Vim将显示类似`foo^@bar`的诡异信息。当你对字符串使用`echom`而不是`echo`时，Vim将输出字符串中额外的字符，这意味着有时`echom`的输出跟`echo`的会有所不同。`^@`在Vim里表示"换行符"。

字符串字面量

Vim也允许你使用"字符串字面量"(译注：如Python中的r"")来避免转义字符串的滥用。 执行下面命令：

```
:::vim
:echom '\n\\'
```

Vim显示`\n\\`。使用单引号将告诉Vim，你希望字符串所见即所得，无视转义字符串。 一个例外是一行中连续两个单引号将产生一个单引号。(译注：相当于某种转义)试试这个命令：

```
:::vim
:echom 'That''s enough.'
```

Vim将显示`That's enough.`。两个单引号是字符串字面量里唯一的特殊序列。

在本书稍后的部分，我们将重新审视字符串字面量的更多内容。(那时我们会深陷于正则表达式)

真值(Truthiness)

你可能想知道Vim怎么对待用在`if`语句中的字符串。执行下面的命令：

```
:::vim
:if "foo"
:  echo "yes"
:else
:  echo "no"
:endif
```

Vim将显示 `no` 。如果你搞不懂为何如此，你应该重读关于条件语句的那一章(译注：第21章)， 因为我们在那里已经讨论过了。

练习

阅读 `:help expr-quote` 。查看在一般的Vim字符串中允许的转义字符串列表。找出如何插入一个tab字符。

尝试找出一种方法，不使用转义字符串来插入一个tab字符。提示：阅读 `:help i_CTRL-V` 。

阅读 `:help literal-string` 。

字符串函数

Vim有许多内置(built-in)函数来操作字符串。本章中我们将介绍一些最为重要的字符串函数。

长度

第一个介绍的函数是 `strlen` 。执行下面的命令：

```
:::vim
:echom strlen("foo")
```

Vim显示 `3` ，也即 `"foo"` 的长度。现在尝试下面的命令：

```
:::vim
:echom len("foo")
```

Vim再一次显示 `3` 。对于字符串， `len` 和 `strlen` 有同样的效果。在本书稍后的章节我们会回过头来探讨 `len` 。

切割

执行下面的命令(注意是 `echo` 而不是 `echom`)：

```
:::vim
:echo split("one two three")
```

Vim显示 `['one','two','three']` 。 `split` 函数把字符串切割成列表。我们将简要介绍列表，但现在不要纠结于此。

你也可以指定一个分隔符来代替"空白"。

```
:::vim
:echo split("one,two,three", ",")
```

Vim再一次显示 `['one','two','three']` ，因为 `split` 的第二个参数是","，表示以","切割。

连接

你不仅可以切割字符串，还可以连接它们。执行下面命令：

```
:::vim
:echo join(["foo", "bar"], "...")
```

Vim将显示 `foo...bar` 。暂时不要在意列表语法。

`split` 和 `join` 可以合作无间。执行下面的命令：

```
:::vim
:echo join(split("foo bar"), ";")
```

Vim显示 `foo;bar` 。首先我们把 `"foo bar"` 切割成列表，接着用分号作为分隔符把列表连接成字符串。

大小写转换

Vim有两个函数来转换字符串大小写。执行下面的命令：

```
:::vim
:echom tolower("Foo")
:echom toupper("Foo")
```

Vim显示 `foo` 和 `FOO` 。这很浅显易懂吧。

在许多语言(如Python)有一个惯例是在进行比较之前把字符串强制转换成小写来实现大小写无关的比较。 在Vimscript里不需要这么做，因为有大小写不敏感的比较运算符。如果你回忆不起来，重新读关于比较的那一章。 (译注：如果你回忆不起来，那是第22章)

你可以自由选择使用 `tolower` 或 `==#` 以及 `==?` 来实现大小写敏感的比较。Vimscript社区对此还没有明显的偏好。 选定一个并在你所有的脚本中保持一致。

练习

执行 `:echo split('1 2')` 和 `:echo split('1,,,2',',')` 。它们表现一致吗？

阅读 `:help splt()` 。

阅读 `:help join()` 。

阅读`:help functions`并浏览有关String的内置函数。使用`/`命令来辅助你(记住，Vim的帮助文件可以以一般文件的方式浏览)。 这里有着许多函数，所以不要纠结于每一个函数的文档。给自己留下印象，以便于将来的不时之用， 这就够了。

Execute命令

`execute` 命令用来把一个字符串当作Vimscript命令执行。在前面的章节我们曾经跟它打过交道， 不过随着对Vimscript中的字符串有更深入的了解，现在我们将再次认识它。

`execute` 基本用法

执行下面的命令：

```
:::vim
:execute "echom 'Hello, world!'"
```

Vim把 `echom 'Hello, world!'` 当作一个命令，而且尽职地在把它输出的同时将消息记录下来。 Execute是一个非常强大的工具，因为它允许你用任意字符串来创造命令。

让我们试试一个更实用的例子。先在Vim里打开一个文件作为准备工作，接着使用 `:edit foo.txt` 在同一个窗口创建新的缓冲区。 现在执行下面的命令：

```
:::vim
:execute "rightbelow vsplit " . bufname("#")
```

Vim将在第二个文件的右边打开第一个文件的竖直分割窗口(vertical split)。为什么会这样？

首先，Vim将 `"rightbelow vsplit"` 和 `bufname('#')` 调用的结果连接在一起，创建一个字符串作为命令。

我们过一段时间才会讲到相应的函数，现在姑且认为它返回前一个缓冲区的路径名。 你可以用 `echom` 来确认这一点。

待 `bufname` 执行完毕，Vim将结果连接成 `"rightbelow vsplit bar.txt"` 。 `execute` 命令将此作为Vimscript命令执行，在新的分割里打开该文件。

Execute危险吗？

在大多数编程语言中使用诸如"eval"来构造可执行的字符串是会受到谴责的(如果不会是更严重的后果)。 因为两个原因，Vimscript中的 `execute` 命令能免于操这份心。

首先，大多数Vimscript代码仅仅接受唯一的来源——用户的输入。 假设有用户想输入一个古怪的字符串来执行邪恶的命令，无所谓，反正这是他们自己的计算机！ 然而在其他语言里，程序通常得接受来自不可信的用户的输入。Vim是一个特殊的环境， 在此无需担心一般的安全性问题。

第二个原因是因为Vimscript有时候处理问题的方式过于晦涩难懂且稀奇古怪。 这时 `execute` 会是完成任务的最简单，最直白的方法。 在大多数其他语言中，使用"eval"不会省下你多少击键的生命，但在Vimscript里这样做可以化繁为简。

练习

浏览 `:help execute` 来明了哪些命令你可以用 `execute` 实现而哪些不可以。 但当涉猎，因为我们很快将重新审视这个问题。

阅读 `:help leftabove` ， `:help rightbelow` ， `:help :split` 和 `:help :vsplit` (注意最后两个条目中额外的分号)。

在你的 `~/.vimrc` 中加入能在选定的分割(竖直或水平，上/下/左/右方位)中打开前一个缓冲区的映射。

Normal命令

目前为止我们已经介绍了几个最为常用的Vimscript命令，但都跟日常中在normal模式下处理文本的方式无关。 有没有一种办法能把我们的脚本跟日常的文本编辑命令结合起来呢？

答案显然是肯定的。之前我们已经见过`normal`命令，是时候更详细地介绍它了。 执行下面的命令：

```
:::vim
:normal G
```

Vim将把你的光标移到当前文件的最后一行，就像是在normal模式里按下`G`。现在执行下面命令：

```
:::vim
:normal ggdd
```

Vim将移动到文件的第一行(`gg`)并删除它(`dd`)。

`normal`命令简单地接受一串键值并当作是在normal模式下输入的。就是那么简单！

避免映射

执行下面的命令来映射`G`键到别的东西：

```
:::vim
:nnoremap G dd
```

现在在normal模式按下`G`将删除一整行。试试这个命令：

```
:::vim
:normal G
```

Vim将删除当前行。`normal`命令将顾及当前的所有映射。

这意味着我们需要给`normal`提供类似于`nnoremap`之于`nmap`的版本， 否则我们没法使用它——考虑到我们猜测不了用户的映射方式。

幸好Vim真的有这样的命令叫`normal!`。执行这个命令：

```
:::vim
:normal! G
```

这次Vim将移动光标到文件底部，即使 `G` 已经被映射了。

在写Vim脚本时，你应该总是使用 `normal!` ，永不使用 `normal` 。不要信任用户在 `~/.vimrc` 中的映射。

特殊字符

如果你使用 `normal!` 一段时间了，就很可能注意到一个问题。试试下面的命令：

```
:::vim
:normal! /foo<cr>
```

第一眼看上去它应该会开始搜索 `foo` ，但你将看到它不会正常工作。 问题在于 `normal!` 不会解析像 `<cr>` 那样的特殊字符序列。

于是，Vim认为你想要搜索字符串序列"foo"，没有意识到你甚至按下了回车来进行搜索！ (译注：原文为you even pressed return to perform the search! 按后文的意思应该是没有按下return,待问作者) 我们将在下一章讨论如何应对这个问题。

练习

阅读 `:help normal` 。在最后部分，你将获得关于下一章主题的提示。

附加题

如果你还没准备好面对挑战，跳过这一节。如果你够胆，祝你好运！

重温 `:help normal` 关于undo的部分。尝试设计一个删除两行却能单独撤销每次删除的映射。建议从 `nnoremap <leader>d dddd` 开始吧。

这次你并不真的需要 `normal!` (`nnoremap` 就够了)， 但是它揭示了一点：有时阅读一个Vim命令的文档可以激发关于别的内容的奇思妙想。

如果你未尝使用过 `helpgrep` 命令，那就是时候用上它了。阅读 `:help helpgrep` 。 留心关于怎样在匹配内容中浏览的部分。

暂时先别纠结模式(patterns)，我们很快就要谈到它们。 现在只需了解你可以用类似 `foo.*bar` 来查找文档中包括该正则模式的行。

不幸的是， `helpgrep` 会不时给你带来挫折感，因为为了找到某些词，你需要懂得去搜索某些词。 我会帮你省下些无用功，这次你得查找到一种手工修改Vim的撤销序列的方法， 这样你映射的两个删除才能独立地撤销。

在以后你要灵活变通(pragmatic)。有时在你迷惘徘徊的时候，Google一下，你就知道。

执行normal!

既然已经学了 `execute` 和 `normal!` ，我们就可以深入探讨一个Vimscript惯用法。 执行下面的命令：

```
:::vim
:execute "normal! gg/foo\<cr>dd"
```

这将移动到文件的开头，查找 `foo` 的首次出现的地方，并删掉那一行。

之前我们尝试过用 `normal!` 来执行一个搜索命令却无法输入必须的回车来开始进行搜索。 结合 `execute` 和 `normal!` 将解决这个问题。

`execute` 允许你创建命令，因而你能够使用Vim普通的转义字符串来生成你需要的"打不出"的字符。 尝试下面的命令：

```
:::vim
:execute "normal! mqA;\<esc>`q"
```

这个命令做了什么？让我们掰开来讲：

- `:execute "normal! ..."` ：执行命令序列，一如它们是在normal模式下输入的，忽略所有映射， 并替换转义字符串。
- `mq` ：保存当前位置到标记"q"。
- `A` ：移动到当前行的末尾并在最后一个字符后进入insert模式。
- `;` ：我们现在位于insert模式，所以仅仅是写入了一个";"。
- `\<esc>` ：这是一个表示Esc键的转义字符串序列，把我们带离insert模式。
- `` `q `` ：回到标记"q"所在的位置。

看上去有点绕，不过它真的很有用：它在当前行的末尾补上一个分号并保持光标不动。 在写Javascript，C或其他以分号作为语句分隔符的语言时，一旦忘记加上分号，这样的映射将助你一臂之力。

练习

重读 `:help expr-quote` (你之前应该看过)来提醒你怎么用 `execute` 通过转义字符串传递特殊字符给 `normal!` 。

在翻开下一章之前，放下本书休息一下。吃一个三明治或喝一杯咖啡(译注：或者茶！)， 喂一下你的宠物——如果你有的话。

基本的正则表达式

Vim是一个文本编辑器，这意味着大量的Vimscript代码将专注于处理文本。 Vim对正则表达式有着强大的支持，尽管一如既往地也有着一些坑。

把下面的文本打到缓冲区中：

```
:::text
max = 10

print "Starting"

for i in range(max):
    print "Counter:", i

print "Done"
```

这个就是我们将用来测试Vimscript的正则支持的文本。它恰好是Python代码，但不要担心你看不懂Python。 它只是一个例子。

我会假定你懂得基本的正则表达式。如果你不懂， 你应该暂停阅读本书并开始阅读Zed Shaw的Learn Regex the Hard Way。 (译注：暂无中文版，也可选择别的书，或者just Google it) 在你看完后再继续。

高亮

在开始之前，先花点时间讲讲搜索高亮，这样我们可以让匹配的内容更明显。

```
:::vim
:set hlsearch incsearch
```

`hlsearch` 让Vim高亮文件中所有匹配项， `incsearch` 则令Vim在你正打着搜索内容时就高亮下一个匹配项

搜索

移动你的光标到文件顶部并执行下面命令：

```
:::vim
/print
```

当你逐字母敲打时，Vim开始在第一行高亮它们。当你按下回车来进行搜索时， 高亮所有的 `print` ， 同时移动你的光标到下一处匹配。

现在尝试执行下面的命令：

```
:::vim
:execute "normal! gg/print\<cr>"
```

这将移动到文件顶部并开始搜索 `print` ， 带我们到第一处匹配。 用的是我们前一章看过的 `:execute "normal! ..."` 语法。

要到达文件中的第二处匹配，你仅需在命令的结尾加一点别的。执行这个命令：

```
:::vim
:execute "normal! gg/print\<cr>n"
```

Vim将移动光标到缓冲区中的第二个 `print` (同时高亮所有匹配)。

让我们尝试从反方向开始。执行这个命令：

```
:::vim
:execute "normal! G?print\<cr>"
```

这次我们用 `G` 移动到文件结尾并用 `?` 来反向搜索。

所有的搜索命令应该已经烂熟于心 —— 我们在让你习惯 `:execute "normal! ..."` 惯用法时已经反复练习过， 因为它让你在Vimscript代码中能够做日常在Vim里做的事。

魔力(Magic)

`/` 和 `?` 命令能接受正则表达式，而不仅仅是普通字符。执行下面命令：

```
:::vim
:execute "normal! gg/for .+ in .+:\<cr>"
```

Vim抱怨说找不到模式！我告诉过你Vim支持正则搜索，所以为何如此？试试下面命令：

```
:::vim
:execute "normal! gg/for .\\+ in .\\+:\<cr>"
```

这次Vim高亮"for"循环，如我们一开始所指望的。在继续阅读之前，花一分钟来想想为何如此。 记住 `execute` 接受一个字符串。

答案在此：我们需要这样写命令的原因有二：

- 首先， `execute` 接受一个字符串，在调用 `normal!` 命令时，双反斜杠将转换成单反斜杠。
- Vim有四种不同的解析正则表达式的"模式"！ 默认模式下需要在 `+` 前加上一个反斜杠来让它表示"一或多个之前的字符"而不是"一个字面意义上的加号"。

直接在Vim里执行搜索，你很容易就注意到它们的不同，输入下面的命令并按下回车：

```
:::vim
/print .\+
```

现在你可以看到 `\+` 的魔力了。双反斜杠仅仅在把模式作为字符串传递给 `execute` 时才需要。

字面量字符串

正如我们在字符串那一章提到的，Vim允许你使用单引号来定义可以直接传递字符的字面量字符串。 比如，字符串 `'a\nb'` 有四个字符长。

我们可以使用字面量字符串来避免频繁敲打双重反斜杠吗？ 先思考这个问题一两分钟，毕竟答案恐怕比你所认为的要更复杂一些。

试试执行下面的命令(注意这次的单引号和单反斜杠)：

```
:::vim
:execute 'normal! gg/for .\+ in .\+:\<cr>'
```

Vim带你到文件的顶部却不再移动到第一个匹配的地方。你猜对了吗？

命令之所以不能工作，是因为我们需要模式中的 `\<cr>` 被转义成回车，来启动搜索。 因为我们用的是字面量字符串，它并不等价于平常在Vim里键入 `/for .\+ in .\+:\<cr>` ， 显然这是无法工作的。

别怕，方法还是比困难多！不要忘了Vim允许字符串连接，所以可以将命令分割成容易理解的一小段。 执行下面的命令：

```
:::vim
:execute "normal! gg" . '/for .\+ in .\+:' . "\<cr>"
```

这种方法可以在传递给 `execute` 之前把三小段字符串连接起来， 而且我们可以为正则使用字面量字符串并为其他的使用一般的字符串。

更多的魔力(Very Magic)

你可能会好奇Vimscript的四种不同的正则解析模式和它们跟Python，Perl或Ruby中的正则表达式有何不同。 你可以阅读它们的文档，如果你乐意。不过如果你只想找到一种简单科学的解决办法，请继续读下去。

执行下面的命令：

```
:::vim
:execute "normal! gg" . '/\vfor .+ in .+:' . "\<cr>"
```

我们又一次把正则表达式放在单独的字面量字符串里，而这次我们用 `\v` 来引导模式。 这将告诉Vim使用它的"very magic"正则解析模式，而该模式就跟其他语言的非常相似。

如果你以 `\v` 开始你的所有正则表达式，你就不用再纠结Vimscript另外三种疯狂的正则模式了。

练习

认真阅读 `:help magic` 。

阅读 `:help pattern-overview` 来看看Vim支持的正则类型。在看到character classes时停下来。

阅读 `:help match` 。尝试手动执行几次 `:match Error /\v.../` 。

在你的 `~/.vimrc` 文件中加入使用 `match` 来高亮多余的空白为错误的映射。建议使用 `<leader>w` 。

加入另一个映射来清除匹配项(比如 `<leader>W`)。

加入一个normal模式下的会在进行搜索时自动插入 `\v` 的映射。 如果你卡在这个练习上，不要忘了Vim的映射是非常简单的，你只需要告诉它把映射键转换成哪些键。

在你的 `~/.vimrc` 文件中加入 `hlsearch` 和 `incsearch` 选项，随你所欲地设置它。

阅读 `:help nohlsearch` 。注意这是一个命令并且不是 `hlsearch` 的"off mode"。

在你的 `~/.vimrc` 文件中加入消除最后一次搜索的匹配项的高亮的映射。

实例研究：Grep 运算符(Operator)，第一部分

在本章和下一章中，我们将使用Vimscript来实现一个相当复杂的程序。我们将探讨一些闻所未闻的东西， 也将在实战中把之前学过的东西联系起来。

在本实例研究中，遇到不熟悉的内容，你得用 `:help` 弄懂它。如果你只是走马观花，就将所获无多。

Grep

如果你未曾用过 `:grep` ，现在你应该花费一分钟读读 `:help :grep` 和 `:help :make` 。 如果之前没用过quickfix window，阅读 `:help quickfix-window` 。

简明扼要地说： `:grep ...` 将用你给的参数来运行一个外部的grep程序，解析结果，填充quickfix列表， 这样你就能在Vim里面跳转到对应结果。

我们将会添加一个"grep运算符"到任意Vim的内置(或自定义！)的动作中，来选择想要搜索的文本， 让 `:grep` 更容易使用。

用法

在写下每一个有意义的Vimscript程序的第一步，你需要思索一个问题："它会被用户怎么使用呢？"。 尝试构思出一种优雅，简易，符合直觉的调用方法。

这次我会替你把这活干了：

- 我们将创造一个"grep运算符"并绑定到 `<leader>g` 。
- 它将表现得同其他任意Vim运算符一样，还可以加入到组合键(比如 `w` 和 `i{`)中。
- 它将立刻开始搜索并打开quickfix窗口展示结果。
- 它将不会跳到第一个结果，因为当第一个结果不是你想要的时候，这样做会困扰你。

一些你将怎么使用它的用例：

- `<leader>giw` : Grep光标下的词(word)。
- `<leader>giW` : Grep光标下的词的大写形式(WORD)。
- `<leader>gi'` : Grep当前所在的单引号括住的词。
- `viwe<leader>g` : 可视状态下选中一个词并拓展选择范围到下一词，然后Grep。

有很多，很多其他的方法可以用它。看上去它好像需要写很多，很多代码， 但事实上我们只需要实现"运算符"功能然后Vim就会完成剩下的工作。

一个原型

在埋头写下巨量(trickey bits)的Vimscript之前，有一个也许会帮上忙的方法是简化你的目标并实现它， 来推测你最终解决方案可能的"外形"。

让我们简化我们的目标为"创造一个映射来搜索光标下的词"。这有用而且应该更简单，所以我们能更快得到可运行的成果。 目前我们将映射它到 `<leader>g` 。

我们从一个映射骨架开始并逐渐填补它。执行这个命令：

```
:::vim
:nnoremap <leader>g :grep -R something .<cr>
```

如果你阅读过 `:help grep` ，你就能轻易理解这个命令。我们之前也看过许多映射，这里没有什么是新的。

显然我们还没做什么，所以让我们一步步打磨这个映射直到它符合我们的要求。

搜索部分

首先我们需要搜索光标下的词，而不是 `something` 。执行下面的命令：

```
:::vim
:nnoremap <leader>g :grep -R <cword> .<cr>
```

现在试一下。 `<cword>` 是一个Vim的command-line模式的特殊变量， Vim会在执行命令之前把它替换为"光标下面的那个词"。

你可以使用 `<cWORD>` 来得到大写形式(WORD)。执行这个命令：

```
:::vim
:nnoremap <leader>g :grep -R <cWORD> .<cr>
```

现在试试把光标放在诸如 `foo-bar` 的词上面。Vim将grep `foo-bar` 而不是其中的一部分。

我们的搜索部分还有一个问题：如果这里面有什么特殊的shell字符，Vim会毫不犹豫地传递给外部的grep命令。 这样会导致程序崩溃(或更糟：铸成某些大错)。

让我们看看如何使它挂掉。输入 `foo;ls` 并把光标放上去执行映射。grep命令失败了， 而Vim将执行 `ls` 命令！这肯定糟透了，如果词里包括比 `ls` 更危险的命令呢？

为了解决这个问题，我们将调用参数用引号括起来。执行这个命令：

```
:::vim
:nnoremap <leader>g :grep -R '<cWORD>' .<cr>
```

大多数shell把单引号括起来的内容当作(大体上)字面量，所以我们的映射现在更加健壮了。

转义Shell命令参数

搜索部分还有一个问题。在 `that's` 上尝试这个映射。它不会工作，因为词里的单引号与grep命令的单引号发生了冲突！

为了解决问题，我们可以使用Vim的 `shellescape` 函数。 阅读 `:help escape()` 和 `:help shellescape()` 来看它是怎样工作的(真的很简单)。

因为 `shellescape()` 要求Vim字符串，我们需要用 `execute` 动态创建命令。 首先执行下面命令来转换 `:grep` 映射到 `:execute "..."` 形式：

```
:::vim
:nnoremap <leader>g :execute "grep -R '<cWORD>' ."<cr>
```

试一下并确信它可以工作。如果不行，找出拼写错误并改正。 然后执行下面的使用了 `shellescape` 的命令。

```
:::vim
:nnoremap <leader>g :execute "grep -R " . shellescape("<cWORD>") . " ."<cr>
```

在一般的词比如 `foo` 上执行这个命令试试。它可以工作。再到一个带单引号的词，比如 `that's` ，上试试看。 它还是不行！为什么会这样？

问题在于Vim在拓展命令行中的特殊变量，比如 `<cWORD>` ， 的之前，就已经执行了 `shellescape()` 。 所以Vim shell-escaped了字面量字符串 `"<cWORD>"` (什么都不做，除了给它添上一对单引号)并连接到我们的 `grep` 命令上。

通过执行下面的命令，你可以亲眼目睹这一切。

```
:::vim
:echom shellescape("<cWORD>")
```

Vim将输出 `'<cWORD>'` 。注意引号也是输出字符串的一部分。Vim把它作为shell命令参数保护了起来。

为解决这个问题，我们将使用 `expand()` 函数来强制拓展 `<cWORD>` 为对应字符串， 抢在它被传递给 `shellescape` 之前。

让我们单独看看这一部分是怎么工作的。把你的光标移到带单引号的词(比如 `that's`)上去，并执行下面命令：

```
:::vim
:echom expand("<cWORD>")
```

Vim输出 `that's` ，因为 `expand("<cWORD>")` 以Vim字符串的形式返回当前光标下的词。 是时候加入 `shellescape` 的部分了：

```
:::vim
:echom shellescape(expand("<cWORD>"))
```

这次Vim输出 `'that'\''s'` 。 如果觉得这看上去真可笑，你大概没有感受过看透了各种shell转义的疯狂形式后的淡定吧。 目前，不用为此而纠结。就相信Vim接受了 `expand` 的输出并正确地转义了它。

目前我们已经得到了光标下的词的彻底转义版本。是时候连接它到我们的映射了！ 执行下面的命令：

```
:::vim
:nnoremap <leader>g :exe "grep -R " . shellescape(expand("<cWORD>")) . " ."<cr>
```

试一下。这个映射不再有问题，即使我们用它搜索带古怪符号的词。

"从简单的Vimscript开始并一点点转变它直到达成你的目标"这样的工作方式将会被你一再取用。

整理整理

在完成映射之前，还要处理一些小问题。首先，我们说过我们不想自动跳到第一个结果， 所以要用 `grep!` 替换掉 `grep` 。执行下面的命令：

```
:::vim
:nnoremap <leader>g :execute "grep! -R " . shellescape(expand("<cWORD>")) . " ."<cr>
```

再一次试试，发现什么都没发生。Vim用结果填充了quickfix窗口，我们却无法打开。 执行下面的命令：

```
:::vim
:nnoremap <leader>g :execute "grep! -R " . shellescape(expand("<cWORD>")) . " ."<cr>:cope
```

现在试试这个映射，你将看到Vim自动打开了包含搜索结果的quickfix窗口。 我们所做的仅仅是在映射的结尾续上 `:copen<cr>` 。

最后一点，在搜索的时候，我们要移除Vim所有的grep输出。执行下面的命令：

```
:::vim
:nnoremap <leader>g :silent execute "grep! -R " . shellescape(expand("<cWORD>")) . " ."<c
```

我们完成了，试一试并犒劳一下自己吧！ `silent` 命令仅仅是在运行一个命令的同时隐藏它的正常输出。

练习

把我们刚刚做出来的映射加入到你的 `~/.vimrc` 文件。

如果你未曾读过 `:help :grep` ，去读它。

阅读 `:help cword` 。

阅读 `:help cnext` 和 `help cprevious` 。修改你的grep映射，试一下它们。

设置 `:cnext` 和 `:cprevious` 的映射，让在匹配内容间的移动更加方便。

阅读 `:help expand` 。

阅读 `:help copen` 。

在我们创建的映射中加入height参数到 `:copen` 命令中，看看quickfix窗口能不能以指定的高度打开。

阅读 `:help silent` 。

实例研究：Grep运算符(Operator)，第二部分

目前为止，我们已经完成了一个原型，是时候扩充它，让它更加强大。

记住：我们初始目标是创建"grep运算符"。我们还需要做一大堆新的东西来达成目标， 但要像前一章的过程一样：从简单的东西开始，并逐步改进直到它满足我们的需求。

在开始之前，注释掉 `~/.vimrc` 中在前一章创建的映射。我们还要用同样的快捷键来映射新的运算符。

新建一个文件

创建一个新的运算符需要许多命令，把它们手工打出来将很快变成一种折磨。 你可以把它附加到 `~/.vimrc` ，但让我们为这个运算符创建一个独立的文件。我们有足够的必要这么做。

首先，找到你的Vim `plugin` 文件夹。在Linux或OS X，这将会是 `~/.vim/plugin` 。 如果你是Windows用户，它将位于你的主目录下的 `vimfiles` 文件夹。(如果你找不到，在Vim里使用`:echo $HOME命令) 如果这个文件夹不存在，创建一个。

在 `plugin/` 下新建文件 `grep-operator.vim` 。这就是你放置新运算符的代码的地方。 一旦文件被修改，你可以执行 `:source %` 来重新加载代码。 每次你打开Vim，这个文件也会被重新加载，就像 `~/.vimrc` 。

不要忘了，在你source之前，你必须先保存文件，这样才能看到变化！

骨架(Skeleton)

要创建一个新的Vim运算符，你需要从两个组件开始：一个函数还有一个映射。 先添加下面的代码到 `grep-operator.vim` ：

```
:::vim
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@

function! GrepOperator(type)
    echom "Test"
endfunction
```

保存文件并用 `:source %` source它。尝试通过按下 `<leader>giw` 来执行"grep整个词"。 Vim将在接受 `iw` 动作(motion)后，输出 `Test` ，意味着我们已经搭起了骨架。

函数部分是简单的，没有什么是我们没讲过的。不过映射部分比较复杂。 我们首先对函数设置了 `operatorfunc` 选项，然后执行 `g@` 来以运算符的方式调用这个函数。 看起来这有点绕，不过这就是Vim工作的原理。

暂时把这个映射看作黑魔法吧。稍后你可以到文档里一探究竟。

可视模式

我们已经在normal模式下加入了这个运算符，但还想要在visual模式下用到它。 在之前的映射下面添加多一个：

```
:::vim
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>
```

保存并source文件。现在在visual模式下选择一些东西并按下 `<leader>g` 。 什么也没发生，但Vim确实输出了 `Test` ，所以我们的函数已经运行了。

之前我们就见过 `<c-u>` ，但是还没有解释它是做什么的。试一下在可视模式下选中一些文本并按下 `:` 。 Vim将打开一个命令行就像平时按下了 `:` 一样，但是命令行的开头自动添加了 `'<,'>` ！

Vim为了提高效率，插入了这些文本来让你的命令在被选择的范围内执行。 但是这次，我们不需要它添倒忙。我们用 `<c-u>` 来执行"从光标所在处删除到行首的内容"，移除多余文本。 最后剩下一个孤零零的 `:` ，为调用 `call` 命令作准备。

我们传递过去的 `visualMode()` 参数还没有讲过呢。 这个函数是Vim的内置函数，它返回一个单字符的字符串来表示visual模式的类型： `"v"` 代表字符宽度(characterwise)， `"V"` 代表行宽度(linewise)， `Ctrl-v` 代表块宽度(blockwise)。

动作类型

我们定义的函数接受一个 `type` 参数。我们知道在visual模式下它将会是 `visualmode()` 的返回值， 但是在normal模式下呢？

编辑函数体部分，让代码像这样：

```
:::vim
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
    echom a:type
endfunction
```

Source文件，然后继续并用多种的方式测试它。你可能会得到类似下面的结果：

- 按下 `viw<leader>g` 显示 `v` ，因为我们处于字符宽度的visual模式。
- 按下 `Vjj<leader>g` 显示 `V` ，因为我们处于行宽度的visual模式。
- 按下 `<leader>giw` 显示 `char` ，因为我们在字符宽度的动作(characterwise motion)中使用该运算符。
- 按下 `<leader>gG` 显示 `line` ，因为我们在行宽度的动作(linewise motion)中使用该运算符。

现在我们已经知道怎么区分不同种类的动作，这对于我们选择需要搜索的词是很重要的。

复制文本

我们的函数将需要获取用户想要搜索的文本，而这样做最简单的方法就是复制它。 把函数修改成这样：

```
:::vim
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
    if a:type ==# 'v'
        execute "normal! `<v`>y"
    elseif a:type ==# 'char'
        execute "normal! `[v`]y"
    else
        return
    endif

    echom @@
endfunction
```

哇。好多新的东西啊。试试按下 `<leader>giw` ， `<leader>g2e` 和 `vi(<leader>g` 看看。 每次Vim都会输出动作所包括的文本，显然我们已经走上正道了！

让我们把这段代码一步步分开来看。首先我们用 `if` 语句检查 `a:type` 参数。如果是 `'v'` ， 它就是使用在字符宽度的visual模式下，所以我们复制了可视模式下的选中文本。

注意我们使用大小写敏感比较 `==#` 。如果我们只用了 `==` 而用户设置 `ignorecase` ， `"V"` 也会是匹配的，结果不会如我们所愿。重视防御性编程！

`if` 语句的第二个分支则会拦住normal模式下使用字符宽度的动作。

剩下的情况只是默默地退出。我们直接忽略行宽度/块宽度的visual模式和对应的动作类型。Grep默认情况下不会搜索多行文本，所以在搜索内容中夹杂着换行符是毫无意义的。

我们每一个 `if` 分支都会执行 `normal!` 命令来做两件事：

- 在可视状态下选中我们想要的文本范围：
  - 先移动到范围开头，并标记
  - 进入字符宽度的visual模式
  - 移动到范围结尾的标记
- 复制可视状态下选中的文本。

先不要纠结于特殊标记方式。你将会在完成本章结尾的练习时学到为什么它们会不一样。

函数的最后一行输出变量 `@@` 。不要忘了以 `@` 开头的变量是寄存器。 `@@` 是"未命名"(unnamed)寄存器： 如果你在删除或复制文本时没有指定一个寄存器，Vim就会把文本放在这里。

简明扼要地说：我们选中要搜索的文本，复制它，然后输出被复制的文本。

转义搜索文本

既然得到了Vim字符串形式的需要的文本，我们可以像前一章一样将它转义。修改 `echom` 命令成这样：

```
:::vim
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
    if a:type ==# 'v'
        normal! `<v`>y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    echom shellescape(@@)
endfunction
```

保存并source文件，然后在可视模式下选中带特殊字符的文本，按下 `<leader>g` 。 Vim显示一个被转义了的能安全地传递给shell命令的文本。

执行Grep

我们终于可以加上 `grep!` 命令来实现真正的搜索。替换掉 `echom` 那一行，代码看起来就像这样：

```
:::vim
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
    if a:type ==# 'v'
        normal! `<v`>y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    silent execute "grep! -R " . shellescape(@@) . " ."
    copen
endfunction
```

看起来眼熟吧。我们简单地执行上一章得到的 `silent execute "grep! ..."` 命令。 由于我们不再把所有的代码塞进单个 `nnoremap` 命令里，现在代码甚至更加清晰易懂了！

保存并source文件，然后尝试一下，享受自己辛勤劳动的成果吧！

因为定义了一个全新的Vim运算符，现在我们可以在许多场景下使用它了，比如：

- `viw<leader>g`：可视模式下选中一个词，然后grep它。
- `<leader>g4w`：Grep接下来的四个词。
- `<leader>gt;`：Grep到分号为止的文本。
- `<leader>gi[`：Grep方括号里的文本.

这里彰显了Vim的优越性：它的编辑命令就像一门语言。当你加入新的动词，它会自动地跟(大多数)现存的名词和形容词搭配起来。

练习

阅读 `:help visualmode()` 。

阅读 `:help c_ctrl-u` 。

阅读 `:help operatorfunc` 。

阅读 `:help map-operator` 。

实例研究：Grep运算符(Operator)，第三部分

我们新鲜出炉的"grep运算符"工作得很好，但是写Vimscript的目的，就是要体贴地改善你的用户的生活。 我们可以额外做两件事，让我们的运算符更加符合Vim生态圈的要求。

保护寄存器

由于把文本复制到未命名寄存器中，我们破坏了之前在那里的内容。

这并不是我们的用户想要的，所以让我们在复制之前先保存寄存器中的内容并于最后重新加载。 修改代码成这样：

```
:::vim
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
    let saved_unnamed_register = @@

    if a:type ==# 'v'
        normal! `<v`>y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    silent execute "grep! -R " . shellescape(@@) . " ."
    copen

    let @@ = saved_unnamed_register
endfunction
```

我们在函数的开头和结尾加入了两个 `let` 语句。 第一个用一个变量保存 `@@` 中的内容，第二个则重新加载保存的内容。

保存并source文件。测试一下，复制一些文本，接着按下 `<leader>giw` 来执行运算符， 然后按下 `p` 来粘贴之前复制的文本。

当写Vim插件时，你总是应该尽量在修改之前保存原来的设置和寄存器值，并在之后加载回去。 这样你就避免了让用户陷入恐慌的可能。

命名空间

我们的脚本在全局命名空间中创建了函数 `GrepOperator` 。 这大概不算什么大问题，但当你写Vimscript的时候，事前以免万一远好过事后万分歉意。

仅需增加几行代码，我们就能避免污染全局命名空间。把代码修改成这样：

```
:::vim
nnoremap <leader>g :set operatorfunc=<SID>GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call <SID>GrepOperator(visualmode())<cr>

function! s:GrepOperator(type)
    let saved_unnamed_register = @@

    if a:type ==# 'v'
        normal! `<v`>y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    silent execute "grep! -R " . shellescape(@@) . " ."
    copen

    let @@ = saved_unnamed_register
endfunction
```

脚本的前三行已经被改变了。首先，我们在函数名前增加前缀 `s:` ，这样它就会处于当前脚本的命名空间。

我们也修改了映射，在 `GrepOperator` 前面添上 `<SID>` ，所以Vim才能找到这个函数。 如果我们不这样做，Vim会尝试在全局命名空间查找该函数，这是不可能找到的。

欢呼吧，我们的 `grep-operator.vim` 脚本不仅非常有用，而且是一个善解人意的Vimscript公民！

练习

阅读 `:help <SID>` 。

享受一下，吃点零食犒劳自己。

列表

目前为止我们已经很熟悉Vimscript里的变量类型了，但我们压根还没讲到集合(aggregates)呢！ Vim有两种主要的集合类型，现在我们将讲到第一种：列表。

Vimscript列表是有序的，异质的元素集合。执行下面的命令：

```
:::vim
:echo ['foo', 3, 'bar']
```

Vim输出这个列表。列表里当然可以嵌套列表。执行下面的命令：

```
:::vim
:echo ['foo', [3, 'bar']]
```

Vim会愉快地输出这个列表。

索引

Vimscript列表的索引从0开始，你可以用下标得到对应元素。执行这个命令：

```
:::vim
:echo [0, [1, 2]][1]
```

Vim显示 `[1,2]` 。你也可以从列表结尾进行索引，这很像Python。执行这个命令：

```
:::vim
:echo [0, [1, 2]][-2]
```

Vim显示 `0` 。索引 `-1` 对应列表的最后一个元素， `-2` 对应倒数第二个，以此类推。

切割

Vim列表也可被切割。这看上去会让Python程序员感到眼熟，但它不总是表现得跟Python中的一样！ 执行这个命令：

```
:::vim
:echo ['a', 'b', 'c', 'd', 'e'][0:2]
```

Vim显示 `['a','b','c']` (第1,2,3个元素)。越过列表索引上界也是安全的。试试这个命令：

```
:::vim
:echo ['a', 'b', 'c', 'd', 'e'][0:100000]
```

Vim仅仅显示整个列表。

可以用负数切割。试试这个命令：

```
:::vim
:echo ['a', 'b', 'c', 'd', 'e'][-2:-1]
```

Vim显示 `['d','e']` (元素-2和-1)。

你可以忽略第一个索引以表示"开头"和/或最后一个索引以表示"结尾"。执行下面的命令：

```
:::vim
:echo ['a', 'b', 'c', 'd', 'e'][:1]
:echo ['a', 'b', 'c', 'd', 'e'][3:]
```

Vim显示 `['a','b']` 和 `['d','e']` 。

像Python，Vimscript也允许你索引和切割字符串。执行下面命令：

```
:::vim
:echo "abcd"[0:2]
```

Vim显示 `abc` 。可是，你不能使用负数来索引字符串。你却可以使用负数切割字符串！ 执行下面的命令：

```
:::vim
:echo "abcd"[-1] . "abcd"[-2:]
```

Vim显示 `cd` (使用负数索引会没有报错地得到一个空字符串)。

连接

你可以用 `+` 连接Vim列表。试试这个命令：

```
:::vim
:echo ['a', 'b'] + ['c']
```

Vim，一点也不奇怪地，显示 `['a','b','c']` 。 没什么好说的——在奇怪的Vimscript世界里，列表是如此地正常，以至于让人感到奇怪。

列表函数

Vim有着许许多多内置列表函数。执行这个命令：

```
:::vim
:let foo = ['a']
:call add(foo, 'b')
:echo foo
```

Vim就地在列表 `foo` 末尾加上 `'b'` ，并显示 `['a','b']` 。

```
:::vim
:echo len(foo)
```

Vim显示 `2` ，即是列表的长度。试试下面的命令：

```
:::vim
:echo get(foo, 0, 'default')
:echo get(foo, 100, 'default')
```

Vim显示 `a` 和 `default` 。 `get` 函数会返回给定索引对应的给定列表的项，如果索引超过列表范围， 返回给定的默认值。

执行这个命令：

```
:::vim
:echo index(foo, 'b')
:echo index(foo, 'nope')
```

Vim显示 `1` 和 `-1` 。 `index` 函数返回给定项在给定列表的第一个索引，如果不在列表中则返回 `-1` 。

现在执行这个命令：

```
:::vim
:echo join(foo)
:echo join(foo, '---')
:echo join([1, 2, 3], '')
```

Vim显示`a b`,`a---b`,和`123`。`join`先将给定列表的每一项强制转换成字符串， 再以给定的分割字符串(或一个空格，如果没有给的话)作为分割，连接成一个字符串。

执行下面命令：

```
:::vim
:call reverse(foo)
:echo foo
:call reverse(foo)
:echo foo
```

Vim先显示`['b','a']`，接着是`['a','b']`。`reverse`就地转置给定的列表。

练习

阅读`:help List`。看完它。注意大写`L`。

阅读`:help add()`.

阅读`:help len()`.

阅读`:help get()`.

阅读`:help index()`.

阅读`:help join()`.

阅读`:help reverse()`.

浏览`:help functions`来查找我没有讲过的其他列表函数。 执行`:match Keyword /\clist/`来以大小写不敏感的方式高亮单词`list`，以便于你的查找。

循环

你可能会惊讶地发现，作为一本关于编程语言的书，在前35章里我们压根就没有提到循环！Vimscript提供了非常多的方式操作文本(比如， `normal!`)， 因此循环并不像在其他大多数语言中的那么必要。

即使如此，总有一天你会需要用到它的，所以现在让我们探讨Vim支持的两种主要的循环。

For循环

第一种循环是 `for` 循环。如果你习惯了Java,C或Javascript中的 `for` 循环，它看上去有点古怪。 但是你会发现这种写法十分地优雅。执行下面的命令：

```
:::vim
:let c = 0

:for i in [1, 2, 3, 4]
:  let c += i
:endfor

:echom c
```

Vim显示 `10` ，就是把列表中的每一个元素的加起来的结果。Vimscript的 `for` 循环遍历整个列表 (或我们待会会提到的字典)。

Vimscript中不存在C风格的 `for (int i = 0; i < foo; i++)` 。这一开始可能难以适应， 但一旦习惯你就不会再怀念C风格的for循环了。

While循环

Vim也支持经典的 `while` 循环。执行下面命令：

```
:::vim
:let c = 1
:let total = 0

:while c <= 4
:  let total += c
:  let c += 1
:endwhile

:echom total
```

Vim再次显示 `10` 。几乎每一个程序猿都熟悉这个循环，所以我们不会浪费时间讲解。 你将会很少用到它。铭记它以备不时之需。

练习

阅读 `:help for` .

阅读 `:help while` .

字典

我们讲到的最后一种Vimscript类型将是字典。 Vimscript字典类似于Python中的dict，Ruby中的hash，和Javascript中的object。

字典用花括号创建。值是异质的，但键会被强制转换成字符串。就是这么简单，你没想到吧？

执行这个命令：

```
:::vim
:echo {'a': 1, 100: 'foo'}
```

Vim显示`{'a':1,'100':'foo'}`，这说明Vimscript的确把键强制转换为字符串，同时保留值不变。

Vimscript避免了Javascript标准的蠢笨之处，允许你在字典的最后一个元素后留下一个逗号。(译注：在Javascript的标准中，最后一个元素后面不能留下一个逗号。 但在Firefox里，留下那个逗号是允许的，不过这是Firefox的问题。) 执行下面的命令：

```
:::vim
:echo {'a': 1, 100: 'foo',}
```

Vim再次显示`{'a':1,'100':'foo'}`(译注：结尾小逗号不见了)。你应该总是在字典里留下一个多余的逗号， 尤其是当字典的定义跨越多行的时候，这样增加新项的时候将不容易犯错。

索引

查找字典中的一个值的语法跟大多数语言是一样的。执行这个命令：

```
:::vim
:echo {'a': 1, 100: 'foo',}['a']
```

Vim显示`1`。试试使用不是字符串的索引：

```
:::vim
:echo {'a': 1, 100: 'foo',}[100]
```

Vim会在查找之前把索引强制转换成字符串，因为键只能是字符串，这么做是合理的。

当键仅由字母，数字和/或下划线组成时，Vimscript也支持Javascript风格的"点"查找。 试试下面的命令：

```
:::vim
:echo {'a': 1, 100: 'foo',}.a
:echo {'a': 1, 100: 'foo',}.100
```

两种情况下，Vim都显示了正确的元素。使用哪种索引字典的方式取决于你自己的偏好。

赋值和添加

像对待变量一样赋值给字典中的项，就可以在字典中轻松地添加新的项。

```
:::vim
:let foo = {'a': 1}
:let foo.a = 100
:let foo.b = 200
:echo foo
```

Vim显示 `{'a': 100, 'b': 200}` 。赋值和添加一个新项的方式是一样的。

移除项

有两种方法可以移除字典中的项。执行下面的命令：

```
:::vim
:let test = remove(foo, 'a')
:unlet foo.b
:echo foo
:echo test
```

Vim显示 `{}` 和 `100` 。 `remove` 函数将移除给定字典的给定键对应的项，并返回被移除的值。 `unlet` 命令也能移除字典中的项，只是不返回值。

你不能移除字典中不存在的项。试试执行这个命令：

```
:::vim
:unlet foo["asdf"]
```

Vim抛出一个错误。

选择 `remove` 还是 `unlet` 很大程度上取决于个人偏好。如果非要我说，我推荐使用 `remove` ，因为它比 `unlet` 更灵活。 `remove` 可以做任何 `unlet` 能做的事，反过来不成立。 所以选择 `remove` 可以一招鲜，吃遍天。

字典函数

就像列表，Vim有许许多多内置的字典函数。执行下面的命令：

```
:::vim
:echom get({'a': 100}, 'a', 'default')
:echom get({'a': 100}, 'b', 'default')
```

Vim显示 `100` 和 `default` ，如同列表版本的 `get` 函数.

你也可以检查给定字典里是否有给定的键。执行这个命令：

```
:::vim
:echom has_key({'a': 100}, 'a')
:echom has_key({'a': 100}, 'b')
```

Vim显示 `1` 和 `0` 。不要忘了，Vimscript把 `0` 当作假而其他数字则是真。

你可以用 `items` 从一个字典中获取对应的键值对，执行这个命令：

```
:::vim
:echo items({'a': 100, 'b': 200})
```

Vim将显示 `[['a',100],['b',200]]` 这样的嵌套列表。到目前为止，Vimscript字典不一定是有序的， 所以不要指望 `items` 的返回结果是有序的！

你可以用 `keys` 返回字典的所有的键和 `values` 返回所有的值。它们的作用一如其名——你可以查一下。

练习

阅读 `:help Dictionary` 。看完它。注意大写 `D` 。

阅读 `:help get()` .

阅读 `:help has_key()` .

阅读 `:help items()` .

阅读 `:help keys()` .

阅读 `:help values()` .

切换

在开头前几章我们曾讲过怎么在Vim里设置选项。 对于布尔选项，我们可以使用 `set someoption!` 来"切换"选项。 如果我们能给这个命令创建一个映射，那就再好不过了。

执行下面的命令：

```
:::vim
:nnoremap <leader>N :setlocal number!<cr>
```

在normal模式中按下 `<leader>N` 看看。Vim将会在开启和关闭行号显示之间切换。 像这样的"切换"映射是十分方便的，因此我们就不需要两个独立的键来开/关。

不幸的是，这只对布尔选项起作用。如果我们想要切换一个非布尔选项，还需要做更多的工作。

切换选项

从创建一个可以切换选项的函数，以及调用该函数的映射开始吧。 把下面的代码加入到你的 `~/.vimrc` (或一个 `~/.vim/plugin/` 中的独立文件，如果你想要的话)：

```
:::vim
nnoremap <leader>f :call FoldColumnToggle()<cr>

function! FoldColumnToggle()
    echom &foldcolumn
endfunction
```

保存并source文件，然后按下 `<leader>f` 试试看。Vim显示当前 `foldcolumn` 选项的值。 如果你不熟悉这个选项，阅读 `:help foldcolumn` 再继续。

让我们添加真正的切换功能。修改代码成这样：

```
:::vim
nnoremap <leader>f :call FoldColumnToggle()<cr>

function! FoldColumnToggle()
    if &foldcolumn
        setlocal foldcolumn=0
    else
        setlocal foldcolumn=4
    endif
endfunction
```

保存并source文件，然后试试看。每次你按下它Vim将显示或隐藏折叠状态条(fold column)。

`if` 语句判断 `&foldcolumn` 是否为真(记住Vim把0看作假而其他数字为真)。 如果是，把它设成0(隐藏它)。否则就设置它为4。就是这么简单。

你可以使用一个简单的函数像这样来切换任何以 `0` 代表关，以其他数字代表开的选项。

切换其他东西

我们的梦想不应止于切换选项。还有一个我们想切换的东西是quickfix窗口。 依然以之前的骨架代码作为起点。加入下面的代码到你的文件：

```
:::vim
nnoremap <leader>q :call QuickfixToggle()<cr>

function! QuickfixToggle()
    return
endfunction
```

这个映射暂时什么都不干。让我们把它转变成其他稍微有点用的东西(不过还没有彻底完成)。把代码改成这样：

```
:::vim
nnoremap <leader>q :call QuickfixToggle()<cr>

function! QuickfixToggle()
    copen
endfunction
```

保存并source文件。如果现在你试一下这个映射，你就会看到一个空荡荡的quickfix窗口。

为了达到实现切换功能的目的，我们将选择一个既快捷又肮脏的手段：全局变量。 把代码改成这样：

```
:::vim
nnoremap <leader>q :call QuickfixToggle()<cr>

function! QuickfixToggle()
    if g:quickfix_is_open
        cclose
        let g:quickfix_is_open = 0
    else
        copen
        let g:quickfix_is_open = 1
    endif
endfunction
```

我们干的事情十分简单 —— 每次调用函数时，我们用一个全局变量来储存quickfix窗口的开关状态。

保存并source文件，接着执行映射试试看。Vim将抱怨变量尚未定义！那么我们先把变量初始化吧。

```
:::vim
nnoremap <leader>q :call QuickfixToggle()<cr>

let g:quickfix_is_open = 0

function! QuickfixToggle()
    if g:quickfix_is_open
        cclose
        let g:quickfix_is_open = 0
    else
        copen
        let g:quickfix_is_open = 1
    endif
endfunction
```

保存并source文件，接着试一下映射。成功了！

改进

我们的切换函数可以工作，但还留有一些问题。

第一个问题是，假设用户用 `:copen` 或 `:cclose` 手动开关窗口，我们的全局变量将不会刷新。实际上这不会是个大问题，因为大多数情况下用户会用这个映射开关窗口，万一没有打开，他们也会再按一次。

这又是关于写Vimscript代码的重要经验：如果你试图处理每一个边际条件，你将陷在里面，而且不会有任何进展。

在大多数情况下，先推出可工作(而且即使不能工作也不会造成破坏)的代码然后回过头改善，要比耗费许多小时苛求完美好得多。除外你正在开发一个很可能有很多人用到的插件。 在这种情况下它才值得耗费时日来达到无懈可击的程度，让用户满意并减少bug报告。

重新加载窗口/缓冲区

我们的函数的另外一个问题是，当用户已经打开了quickfix窗口，并执行这个映射时， Vim关闭了窗口，接着把他们弹到上一个分割中，而不是送他们回之前的地方。 如果你仅仅想快速查看一下quickfix窗口然后继续工作，发生这种事是让人恼怒的。

为了解决这个问题，我们将引入一种写Vim插件时非常有用的惯用法。把你的代码改成这样：

```
:::vim
nnoremap <leader>q :call QuickfixToggle()<cr>

let g:quickfix_is_open = 0

function! QuickfixToggle()
    if g:quickfix_is_open
        cclose
        let g:quickfix_is_open = 0
        execute g:quickfix_return_to_window . "wincmd w"
    else
        let g:quickfix_return_to_window = winnr()
        copen
        let g:quickfix_is_open = 1
    endif
endfunction
```

我们在映射中加入了新的两行。其中一行(在 `else` 分支)设置了另一个全局变量，来保存执行 `:copen` 时的当前窗口序号。

另一行(在 `if` 分支)执行以那个序号作前缀的 `wincmd w` ，来告诉Vim跳转到对应窗口。

我们的解决方法又一次不是无懈可击的，用户可能在两次执行映射之间打开或关闭新的分割。 即使这样，它还是适合于大多数场合，所以目前这已经够好的了。

在大多数程序中，这种手工保存全局状态的伎俩会遭到谴责，但对于一个非常短小的Vimscript函数而言， 它既快捷又肮脏，却能不辱使命，完成重任。

练习

阅读 `:help foldcolumn` .

阅读 `:help winnr()`

阅读 `:help ctrl-w_w` .

阅读 `:help wincmd` .

在需要的地方加上 `s:` 和 `<SID>` 来把函数限定在独自的命名空间中。

函数式编程

现在让我们小憩一下，聊一聊一种你可能听过的编程风格：函数式编程。

如果你用过Python，Ruby或Javascript，甚或Lisp，Scheme，Clojure或Haskell， 你应该会觉得把函数作为变量类型，用不可变的状态作为数据结构是平常的事。 如果你没用过，你可以放心地跳过这一章了，但我还是鼓励你找机会去试试并拓宽自己的视野。

Vimscript具有使用函数式风格进行编程的潜力，不过会有点吃力。 我们可以创建一些辅助函数来让这个过程少些痛苦。

继续前进并创建 `functional.vim` 文件，这样你就不用反复地重新击打每一行代码。 这个文件将会成为这一章的草稿本。

不可变的数据结构

不幸的是，Vim没有类似于Clojure内置的vector和map那样的不可变集合， 不过通过一些辅助函数，我们可以在一定程度上模拟出来。

在你的文件加上下面的函数:

```
:::vim
function! Sorted(l)
    let new_list = deepcopy(a:l)
    call sort(new_list)
    return new_list
endfunction
```

保存并source文件，然后执行 `:echo Sorted([3,2,4,1])` 来试试看。 Vim输出 `[1,2,3,4]` 。

这跟调用内置的 `sort()` 函数有什么区别呢？关键在于第一行： `let new_list = deepcopy(a:l)` 。 Vim的 `sort()` 就地重排列表，所以我们先创建一个列表的副本，并排序副本, 这样原本的列表不会被改变。

这样就避免了副作用，并帮助我们写出更容易推断和测试的代码。让我们加入更多同样风格的辅助函数：

```
:::vim
function! Reversed(l)
    let new_list = deepcopy(a:l)
    call reverse(new_list)
    return new_list
endfunction

function! Append(l, val)
    let new_list = deepcopy(a:l)
    call add(new_list, a:val)
    return new_list
endfunction

function! Assoc(l, i, val)
    let new_list = deepcopy(a:l)
    let new_list[a:i] = a:val
    return new_list
endfunction

function! Pop(l, i)
    let new_list = deepcopy(a:l)
    call remove(new_list, a:i)
    return new_list
endfunction
```

除了中间的一行和它们接受的参数，每一个函数都是一样的。保存并source文件，在一些列表上试试它们。

`Reversed()` 接受一个列表并返回一个新的倒置了元素的列表。

`Append()` 返回一个在原列表的基础上增加了给定值的新列表。

`Assoc()` ("associate"的缩写)返回一个给定索引上的元素被替换成新值的新列表。

`Pop()` 返回一个给定索引上的元素被移除的新列表。

作为变量的函数

Vimscript支持使用变量储存函数，但是相关的语法有点愚钝。执行下面的命令：

```
:::vim
:let Myfunc = function("Append")
:echo Myfunc([1, 2], 3)
```

Vim意料之中地显示 `[1, 2, 3]` 。注意我们使用的变量以大写字母开头。 如果一个Vimscript变量要引用一个函数，它就要以大写字母开头。

就像其他种类的变量，函数也可以储存在列表里。执行下面命令：

```
:::vim
:let funcs = [function("Append"), function("Pop")]
:echo funcs[1](['a', 'b', 'c'], 1)
```

Vim显示 `['a', 'c']` 。 `funcs` 变量不需要以大写字母开头，因为它储存的是列表，而不是函数。 列表的内容不会造成任何影响。

高阶函数

让我们创建一些用途广泛的高阶函数。如果你需要解释，高阶函数就是接受别的函数并使用它们的函数。

我们将从 `map` 函数开始。在你的文件中添加这个：

```
:::vim
function! Mapped(fn, l)
    let new_list = deepcopy(a:l)
    call map(new_list, string(a:fn) . '(v:val)')
    return new_list
endfunction
```

保存并source文件，执行下面命令试试看：

```
:::vim
:let mylist = [[1, 2], [3, 4]]
:echo Mapped(function("Reversed"), mylist)
```

Vim显示 `[[2, 1], [4, 3]]` ， 正好是对列表中的每一个元素应用了 `Reversed()` 的结果。

`Mapped()` 是如何起作用的？我们又一次用 `deepcopy()` 创建新的列表，修修改改，返回修改后的副本 —— 没什么是新的。有门道的是中间的部分。

`Mapped()` 接受两个参数：一个funcref("储存一个函数的变量"在Vim里的说法)和一个列表。 我们使用内置的 `map()` 函数实现真正的工作。现在就阅读 `:help map()` 来看它怎么工作的。

现在我们将创建一些通用的高阶函数。把下面的代码加入到你的文件：

```
:::vim
function! Filtered(fn, l)
    let new_list = deepcopy(a:l)
    call filter(new_list, string(a:fn) . '(v:val)')
    return new_list
endfunction
```

用下面的命令尝试 `Filtered()` ：

```
:::vim
:let mylist = [[1, 2], [], ['foo'], []]
:echo Filtered(function('len'), mylist)
```

Vim显示 `[[1, 2], ['foo']]` 。

`Filtered()` 接受一个谓词函数和一个列表。它返回一个列表的副本， 而这个列表只包括将自身作为谓词函数的输入参数并返回真值的元素。 这里我们使用了内置的 `len()` ，让它过滤掉所有长度为0的元素。

最后我们创建了 `Filtered()` 的好基友(counterpart)：

```
:::vim
function! Removed(fn, l)
    let new_list = deepcopy(a:l)
    call filter(new_list, '!' . string(a:fn) . '(v:val)')
    return new_list
endfunction
```

像使用 `Filtered()` 一样试一下：

```
:::vim
:let mylist = [[1, 2], [], ['foo'], []]
:echo Removed(function('len'), mylist)
```

Vim显示 `[[], []]` 。 `Removed()` 就像 `Filtered()` ，不过它只保留谓词函数返回非真值的元素。

代码中的唯一不同在于调用命令前面的 `'!' .` ，它把谓词函数的结果取反。

效率

考虑到Vim不得不持续地创建新的副本并垃圾回收旧的对象，你可能会认为不停地制造副本是种浪费。

是的，你是对的！Vim的列表不像Clojure的vector那样支持结构共享(structural sharing)， 所以这里所有的复制操作是昂贵的。

有时这的确是个问题。如果你需要使用庞大的列表，程序就会因此变慢。 在现实世界，你可能会吃惊地发现你几乎不会注意到其中的差别。

想想看吧：当我正写下本章时，Vim占用了80M内存(而且我可是装了一堆插件)。 我的笔记本总共有*8G*内存。有一些列表的副本被创建出来，这会造成可被察觉的不同吗？ 当然这取决于列表的大小，但在大多数情况下答案将会是"No"。

作为比较，我的Firefox打开了五个tab，现在正饕餮着*1.22G*内存。

你将需要自己判断，什么时候这种编程风格会导致不可接受的低效率。

练习

阅读 `:help sort()` 。

阅读 `:help reverse()` 。

阅读 `:help copy()` 。

阅读 `:help deepcopy()` 。

阅读 `:help map()` ，如果你未曾读过。

阅读 `:help function()` 。

修改 `Assoc()` ，`Pop()` ，`Mapped()` ，`Filtered()` 和 `Removed()` 来支持字典类型。 你可能需要阅读 `:help type()` 来帮助自己。

实现 `Reduced()` 。

倒给自己一杯最喜欢的饮料。这一章真激烈(intense)！

路径

Vim是一个文本编辑器，而文本编辑器(经常)处理文本文件。文本文件储存在文件系统中， 而我们使用路径来描述文件。Vimscript有一些内置的方法会在你需要处理路径时帮上大忙。

绝对路径

有时外部脚本也需要获取特定文件的绝对路径名。执行下面的命令：

```
:::vim
:echom expand('%')
:echom expand('%:p')
:echom fnamemodify('foo.txt', ':p')
```

第一个命令显示我们正在编辑的文件的相对路径。 `%` 表示"当前文件"。 Vim也支持其他一些字符串作为 `expand()` 的参数。

第二个命令显示当前文件的完整的绝对路径名。字符串中的 `:p` 告诉Vim你需要绝对路径。 这里也有许多别的修饰符可以用到。

第三个命令显示了当前文件夹下的文件 `foo.txt` 的绝对路径，无论文件是否存在。(译注：试一下看看文件不存在的情况？) `fnamemodify()` 是一个比 `expand()` 灵活多了的Vim函数， 你可以指定任意文件名作为 `fnamemodify()` 的参数，而不仅仅是 `expand()` 所需要的那种特殊字符串。

列出文件

你可能想要得到一个特定文件夹下的文件列表。执行下面的命令：

```
:::vim
:echo globpath('.', '*')
```

Vim将输出当前目录下所有的文件和文件夹。 `globpath()` 函数返回一个字符串， 其中每一项都用换行符隔开。为了得到一个列表，你需要自己去 `split()` 。执行这个命令：

```
:::vim
:echo split(globpath('.', '*'), '\n')
```

这次Vim显示一个包括各个文件路径的Vimscript列表。 如果你的文件名里包括了换行符，那就只能由你自己想办法了。

`globpath()` 的通配符(wildcards)的工作方式就像你所想的一样。执行下面的命令：

```
:::vim
:echo split(globpath('.', '*.txt'), '\n')
```

Vim显示一个当前文件夹下的所有 `.txt` 文件组成的列表。

你可以用 `**` 递归地列出文件。执行这个命令：

```
:::vim
:echo split(globpath('.', '**'), '\n')
```

Vim将列出当前文件夹下的所有文件及文件夹。

`globpath()` 非常地强大。在你完成本章练习后，你将学到更多内容。

练习

阅读 `:help expand()` .

阅读 `:help fnamemodify()` .

阅读 `:help filename-modifiers` .

阅读 `:help simplify()` .

阅读 `:help resolve()` .

阅读 `:help globpath()` .

阅读 `:help wildcards` .

创建一个完整的插件

在前四十来章中，我们讲解了许多基础方面的内容。 在本书的最后部分，我们将尝试从零开始为一门语言创造Vim插件。

这不是个适合懦夫的游戏。这将需要你竭尽全力。

如果你现在就想退出，那确实也不坏！你已经学到了如何在 `~/.vimrc` 里改善你的生活， 还有如果修复别人的插件里的bugs。

有"这就够了，我不想虚掷光阴于创造一个我将不会使用的插件"这种想法并不可耻。 现实一点。如果你不想创造一个自己想用的插件，现在就可以离开，到你想要的时候再回来吧。

如果你真的想要继续，确保你可以挤出一些时间。本书剩余部分将会显得困难， 而且我会假定你真的想学点东西，而不是仅仅慵懒地一章章翻过去。

Potion

我们创造的插件将为Potion这门语言提供支持。

Potion是由 `Why the lucky stiff` 在隐于江湖之前(before his disappearance)创建的一门玩具语言。 它非常的简单，所以我们就拿它一试身手。

Potion跟Io很像，同时又借鉴了Ruby,Lua以及其他语言。如果你未曾玩过Io，它可能看上去略古怪。 我强烈推荐你花上至少一两个小时的时间玩玩Potion。在现实生活中你不会用它， 但是它可能会改变你思考的方式并带给你新的思想。

Potion的当前实现相当地粗糙。举个例子：如果你犯了语法错误，它通常会还你段错误。 不要太纠结于此。我会给你许多可用的代码示范，这样你就能更关注于Vimscript本身而非Potion。

我们的目标不是学习Potion(尽管那也挺有趣)。 我们的目标是以Potion作为一个小例子来体验写一个完整的Vim插件的方方面面。

练习

下载并安装Potion。这个就要你自己动手了。它应该会比较简单的。

确保你可以在Potion解释器和以 `.pn` 文件的形式运行小册子里的第一个示例代码。 如果解释器貌似不能工作，看这个issue来查找可能的原因。

旧社会下的插件配置方式

我们需要讲到的第一件事是如何配置我们的插件。在过去，这会是一次混乱的折腾， 但现在我们有一个工具可以非常方便地安装Vim插件。

我们需要先过一下基本的配置方式，然后我们会讲到如何省下麻烦。

基本配置方式

Vim支持把插件分割成多个文件。你可以在 `~/.vim` 下创建许多不同种类的文件夹来放置不同的内容。

我们现在将讲述其中最为重要的几个文件夹，但不会在上面花费太多时间。 当我们创造Potion插件时，我们会逐一认识它们的。

在我们继续前进之前，需要先确定一些用词规范。

我将用"插件"表示一大堆做一系列相关事情的Vimscript代码。 在Vim里，"插件(plugin)"有一个更专业的定义，它表示" `~/.vim/plugins/` 下的一个文件"。

在大多数时间里，我将使用第一个定义。如果指的是第二个定义，我会特意指明。

~/.vim/colors/

Vim将会查找 `~/.vim/colors/mycolors.vim` 并执行它。 这个文件应该包括生成你的配色方案所需的一切Vimscript命令。

本书中，我们不会谈到配色方案。如果想创造属于自己的配色方案，你应该从一个现存的配色方案上改造出来。 记住， `:help` 将与你常在。

~/.vim/plugin/

`~/.vim/plugin/` 下的文件将在每次Vim启动的时候执行。 这里的文件包括那些无论何时，在启动Vim之后你就想加载的代码。

~/.vim/ftdetect/

`~/.vim/ftdetect/` 下的文件在每次你启动Vim的时候也会执行。

`ftdetect` 是"filetype detection"的缩写。 这里的文件仅仅负责启动检测和设置文件的 `filetype` 类型的自动命令。 这意味着它们一般不会超过一两行。

~/.vim/ftplugin/

`~/.vim/ftplugin/` 下的文件则各不相同。

一切皆取决于它的名字!当Vim把一个缓冲区的 `filetype` 设置成某个值时， 它会去查找 `~/.vim/ftplugin/` 下对应的文件。 比如：如果你执行 `set filetype=derp` ，Vim将查找 `~/.vim/ftplugin/derp.vim` 。 一旦文件存在，Vim将执行它。

Vim也支持在 `~/.vim/ftplugin/` 下放置文件夹。 再以我们刚才的例子为例： `set filetype=derp` 将告诉Vim去执行 `~/.vim/ftplugin/derp/` 下的全部 `*.vim` 文件。 这使得你可以按代码逻辑分割在 `ftplugin` 下的文件。

因为每次在一个缓冲区中执行 `filetype` 时都会执行这些文件，所以它们只能设置buffer-local选项！ 如果在它们中设置了全局选项，所有打开的缓冲区的设置都会遭到覆盖！

~/.vim/indent/

`~/.vim/indent/` 下的文件类似于 `ftplugin` 下的文件。加载时也是只加载名字对应的文件。

`indent` 文件应该设置跟对应文件类型相关的缩进，而且这些设置应该是buffer-local的。

是的，你当然可以把这些代码也一并放入 `ftplugin` 文件， 但最好把它们独立出来，让其他Vim用户理解你的意图。这只是一种惯例，不过请尽量体贴用户并遵从它。

~/.vim/compiler/

`~/.vim/compiler` 下的文件非常类似于 `indent` 文件。它们应该设置同类型名的当前缓冲区下的编译器相关选项。

不要担心不懂什么是"编译器相关选项"。我们等会会解释。

~/.vim/after/

`~/.vim/after` 文件夹有点神奇。这个文件夹下的文件会在每次Vim启动的时候加载， 不过是在 `~/.vim/plugin/` 下的文件加载了之后。

这允许你覆盖Vim的默认设置。实际上你将很少需要这么做，所以不用理它， 除非你有"Vim设置了选项 `x` ，但我想要不同的设置"的主意。

~/.vim/autoload/

`~/.vim/autoload`文件夹就更加神奇了。事实上它的作用没有听起来那么复杂。

简明扼要地说：`autoload`是一种延迟插件代码到需要时才加载的方法。 我们将在重构插件的时候详细讲解并展示它的用法。

~/.vim/doc/

最后，`~/.vim/doc/`文件夹提供了一个你可以放置你的插件的文档的地方。 Vim对文档的要求是多多益善(看看我们执行过的所有`:help`命令就知道)，所以为你的插件写文档是重要的。

练习

重读本章。我没开玩笑。确保你(大体上)明白我们讲过的每一个文件夹。

作为额外的加分，找一些你正在用的Vim插件看看它们如何组织代码文件。

新希望：用Pathogen配置插件

Vim的插件配置方式，在你仅仅添加一个文件来自定义自己的Vim体验时很合理， 但当你想要使用别人写的插件时，这种方式会导致一团糟。

在过去，要想使用别人写好的插件，你得下载所有文件并逐一正确地放置它们。 你也可能使用 `zip` 或 `tar` 来替你做放置的工作。

在这个过程中有些明显的问题：

- 当你想更新插件的时候怎么办？你可以覆盖旧的文件， 但如果作者删除了某个文件，你怎么知道你要手工删除对应文件？
- 假如有两个插件正好使用了同样的文件名(比如 `utils.vim` 或别的更大众的名字)呢？ 有时你可以简单地重命名掉它，但如果它位于 `autoload/` 或别的名字相关的文件夹中呢？ 你改掉文件名，就等于改掉插件。这一点也不好玩。

人们总结出一系列hacks来让事情变得简单些，比如Vimball。 幸运的是，我们不再需要忍受这些肮脏的hacks。 Tim Pope创造了著名的Pathogen插件让管理大量插件变得轻松愉快， 只要插件作者神志清醒地安排好插件结构。(译注：现在推荐vundle来代替Pathogen，前者支持使用git下载插件)

让我们了解一下Pathogen的工作方式，以及为了让我们的插件更加兼容，我们需要做的事。

运行时路径

当Vim在特殊的文件夹，比如 `syntax/` ， 中查找文件时，它不仅仅只到单一的地方上查找。 就像Linux/Unix/BSD系统上的 `PATH` ， Vim设置 `runtimepath` 以便查找要加载的文件。

在你的桌面创建 `colors` 文件夹。在这个文件夹中创建一个叫 `mycolor.vim` 的文件(在本示例中你可以让它空着)。 打开Vim并执行这个命令：

```
:::vim
:color mycolor
```

Vim将显示一个错误，因为它不懂得去你的桌面查找。现在执行这个命令：

```
:::vim
:set runtimepath=/Users/sjl/Desktop
```

当然，你得根据你的情况修改路径名。现在再尝试color命令：

```
:::vim
:color mycolor
```

这次Vim找到了 `mycolor.vim` ，所以它将不再报错。由于文件是空的，它事实上什么都没做，但由于它不再报错，我们确信它找到了。

Pathogen

Pathogen插件在你加载Vim的时候自动地把路径加到你的 `runtimepath` 中。 所有在 `~/.vim/bundle/` 下的文件夹将逐个加入到 `runtimepath` 。(译注：vundle也是这么做的)

这意味着每个 `bundle/` 下的文件夹应该包括部分或全部的标准的Vim插件文件夹，比如 `colors/` 和 `syntax/` 。 现在Vim可以从每个文件夹中加载文件，而且每个插件文件都独立于自己的文件夹中。

这么一来更新插件就轻松多了。你只需要整个移除旧的插件文件夹，并迎来新的版本。 如果你通过版本控制来管理 `~/.vim` 文件夹(你应该这么做)， 你可以使用Mercurial的subrepo或Git的submodule功能来直接签出(checkout)每个插件的代码库， 然后用一个简单的 `hg pull; hg update` 或 `git pull origin master` 来更新。

成为Pathogen兼容的

我们计划让我们的用户通过Pathogen安装我们写的Potion插件。 我们需要做的：在插件的代码库里，放置我们的文件到正确的文件夹中。就是这么简单！

我们插件的代码库展开后看起来就像这样：

```
:::text
potion/
    README
    LICENSE
    doc/
        potion.txt
    ftdetect/
        potion.vim
    ftplugin/
        potion.vim
    syntax/
        potion.vim
    ... etc ...
```

我们把它放置在GitHub或Bitbucket上，这样用户就能简单地clone它到 `bundle/` ，一切顺利！

练习

如果你还没有安装[vnudle][]，安装它。(译注：原文是安装Pathogen，但是没有必要啦)

给你的插件创建Mercurial或Git代码库，起名叫 `potion` 。 你可以把它放到你喜欢的地方，并链接到 `~/.vim/bundle/potion/` 或就把它直接放到 `~/.vim/bindle/potion/` 。

在代码库中创建 `README` 和 `LICENSE` 文件，然后commit。

Push到Bitbucket或GitHub。

阅读 `:help runtimepath` 。

检测文件类型

让我们创建一个Potion文件作为插件的测试样本。

```
:::text
factorial = (n):
    total = 1
    n to 1 (i):
        total *= i.
    total.

10 times (i):
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.
```

这个代码创建了一个简单的阶乘函数并调用它10次，逐次输出结果。继续前进并用 `potion factorial.pn` 执行它。 输出结果应该像这样：

```
:::text
0! is: 0
1! is: 1
2! is: 2
3! is: 6
4! is: 24
5! is: 120
6! is: 720
7! is: 5040
8! is: 40320
9! is: 362880
```

如果你得不到这个输出，或者你得到一个错误，停下来并排查问题所在。 这个代码应该会正常工作的。

这跟学习Vimscript没有关系，不过它能让你成为更好的程序猿。

检测Potion文件

用Vim打开 `factorial.pn` 并执行下面命令：

```
:::vim
:set filetype?
```

Vim将显示 `filetype=` ，因为它还不认识 `.pn` 文件。让我们解决这个问题。

在你的插件的repo中创建 `ftdetect/potion.vim` 。在它里面加入下面几行：

```
:::vim
au BufNewFile,BufRead *.pn set filetype=potion
```

这创建了一个单行自动命令：一个设置 `.pn` 文件的filetype为 `potion` 的命令。很简明吧。

注意我们没有像之前经常做的那样使用一个自动命令组。 Vim自动替你把 `ftdetect/*.vim` 文件包装成自动命令组，所以你不需要操心。

关闭 `factorial.pn` 并重新打开它。现在再执行前面的命令：

```
:::vim
:set filetype?
```

这次Vim显示 `filetype=potion` 。当Vim启动时，它加载 `~/.vim/bundle/potion/ftdetect/potion.vim` 里的自动命令组， 而当它打开 `factorial.pn` 时，自动命令起效，设置 `filetype` 为 `potion` 。

既然已经让Vim识别了Potion文件，我们可以继续前进来做些有用的东西了。

练习

阅读 `:help ft` 。不要担心你看不懂里面的内容。

阅读 `:help setfiletype` 。

修改Potion插件中的 `ftdetect/potion.vim` 。 用 `setfiletype` 代替 `set filetype` 。

基本语法高亮

既然已经移除前进路上的绊脚石，是时候开始为我们的Potion插件写下一些有用的代码。 我们将从一些简单的语法高亮开始。

在你的插件的repo中创建 `syntax/potion.vim` 。把下面的代码放到你的文件里：

```
:::vim
if exists("b:current_syntax")
    finish
endif

echom "Our syntax highlighting code will go here."

let b:current_syntax = "potion"
```

关闭Vim，然后打开你的 `factorial.pn` 文件。 你也许或也许不能看到消息，取决于你是否有其他插件在该插件之后输出消息。 如果你执行 `:message` ， 你将会看到这个文件的确已经加载了。

注意： 每次我告诉你打开Potion文件，我是想要你在一个新的Vim窗口或进程里打开，而不是在一个分割或tab。 打开一个新的Vim窗口导致Vim为此重新加载你所有的插件，而打开一个分割则不会。

代码文件开头和结尾的那几行是一个惯用法，如果这个缓冲区的语法高亮已经启动了，那就无需重新加载。

高亮关键字

在本章的剩下部分，我们将忽略文件开头和结尾的 `if` 和 `let` 防御墙。不要移除那几行，只是眼不见为净而已。

用下面的代码替换掉文件中的占位符 `echom` ：

```
:::vim
syntax keyword potionKeyword to times
highlight link potionKeyword Keyword
```

关闭 `factorial.pn` 并重新打开它。 `to` 和 `times` 被高亮成你的配色方案中的关键字类型了！

这两行展示了Vim中的基本的语法高亮。为了高亮某个语法：

- 你首先要用 `syntax keyword` 或相关命令(我们待会会提到)，定义一组语法类型。
- 然后你要把这组类型链接到高亮组(highlighting groups)。 一个高亮组是你在配色方案里定义的东西，比如"函数名应该是蓝色的"。

这可以让插件作者决定有意义的语法类型分组，然后链接到通用的高亮组。 这同时也让配色方案创作者决定通用的程序结构，而不需要考虑单独的语言。

除了在我们的玩具程序中用到的，Potion还有其他的关键字，所以让我们修改syntax文件来一并高亮它们。

```
:::vim
syntax keyword potionKeyword loop times to while
syntax keyword potionKeyword if elsif else
syntax keyword potionKeyword class return

highlight link potionKeyword Keyword
```

首先要说的是：最后一行没有改掉。我们依然告诉Vim所有在 `potionKeyword` 中的内容应该作为 `Keyword` 高亮。

我们现在新增三行，每行都以 `syntax keyword potionKeyword` 开头。 这意味着多次执行这个命令不会重置语法类型分组 —— 而是扩增它！这使得你可以化整为零地定义分组。

怎样定义分组取决于你：

- 你可以仅仅一行密密麻麻地写满所有的内容。
- 你可以划分成几行，来满足每行80列的规则以便于阅读。
- 你可以每一项都独占一行，来使得diff的结果更加清晰。
- 你可以跟我在这里做的一样，把相关的项放在同一行。

高亮函数

Vim的另一个高亮组是 `Function` 。这就来加入一些Potion的内置函数到我们的高亮文件。 把你的syntax文件修改成这样：

```
:::vim
syntax keyword potionKeyword loop times to while
syntax keyword potionKeyword if elsif else
syntax keyword potionKeyword class return

syntax keyword potionFunction print join string

highlight link potionKeyword Keyword
highlight link potionFunction Function
```

关闭并重新打开 `factorial.pn` ，你将看到内置的Potion函数现在已经高亮了。

它的工作原理就跟关键字高亮一样。我们定义了新的语法类型分组并链接到不同的高亮组。

练习

想一想为什么文件开头的 `if exists` 和结尾的 `let` 是有用的。如果你搞不懂，不要担心。 我也曾就这个问题问过Tim Pope。

浏览 `:help syn-keyword` 。注意提到 `iskeyword` 的部分。

阅读 `:help iskeyword` .

阅读 `:help group-name` 来了解一些配色方案作者常用的通用高亮组。

高级语法高亮

目前我们已经为Potion文件实现了简单的关键字和函数的语法高亮。

如果没有做上一章的练习，你需要回去完成。我将假设你做了练习。

事实上，你应该回去完成你跳过的任何练习。即使你觉得你不需要，为了更好的学习效果，你都得把它们完成了。请在这一点上相信我。

高亮注释

接下来我们需要高亮Potion的一个重要组成部分——注释。 问题是，Potion的注释以 `#` 开头，而 `#` 并不在 `iskeyword` 里。

如果不知道什么是 `iskeyword` ，你没有认真听讲。回去并完成那该死的练习。 在写每一章的内容时，我不会把无意义的粗重活丢给你。你真的需要完成它们来跟上本书的进度。

因为 `#` 不是一个keyword字符，我们需要使用正则表达式来匹配它(以及接下来的注释)。 我们将用 `syntax match` 代替 `syntax keyword` 。在你的syntax文件中加入下面几行：

```
:::vim
syntax match potionComment "\v#.*$"
highlight link potionComment Comment
```

我不会再唠叨要把它们放到文件的哪里。你已经是个程序猿了：由你自己判断。

关闭并重新打开 `factorial.pn` 。在文件的某处添加一个注释，你将看到它被当作注释高亮了。

第二行是简单的：它告诉Vim高亮 `potionComment` 语法类型组里的任何东西为 `Comment` 。

第一行有点新东西。我们使用 `syntax match` 来告诉Vim用正则表达式而不是关键词来匹配。

注意正则表达式以 `\v` 开头，表示使用"very magic"模式。 如果你不太清楚，重读关于基本正则表达式的那一章。(译注：第31章)

当前状况下，"very magic"模式不是必须的。 但将来我们可能会改变这个正则表达式，然后苦思冥想为何它不工作了， 所以我建议总是使用"very magic"来保证一致性。

至于正则表达式的功能，非常简单：匹配以 `#` 开头的注释，包括以此开始到行末的所有字符。

如果你需要重新唤起对正则表达式的记忆，你应该看一下 Zed Shaw的Learn Regex the Hard Way。

高亮运算符

另一个需要正则表达式来高亮的部分是运算符。在你的syntax文件中加入下列内容：

```
:::vim
syntax match potionOperator "\v\*"
syntax match potionOperator "\v/"
syntax match potionOperator "\v\+"
syntax match potionOperator "\v-"
syntax match potionOperator "\v\?"
syntax match potionOperator "\v\*\="
syntax match potionOperator "\v/\="
syntax match potionOperator "\v\+\="
syntax match potionOperator "\v-\="

highlight link potionOperator Operator
```

关闭并重新打开 `factorial.pn` 。注意到阶乘函数的 `*=` 现在被高亮了。

你可能首先注意到，我把每个正则表达式独立成一行而不是像对关键字一样分成组。 这是因为 `syntax match` 不支持在一行里放多个组。

你应该也注意到每个正则表达式都以 `\v` 开头，即使并不是必须的。 在写Vimscript时，我希望保持正则表达式的一致性，即使这样做需要多打几个符号。

你可能会奇怪，为什么我不用类似于 `"\v-\=?"` 的正则表达式来同时匹配 `-` 以及 `-=` 。 你想要的话也可以这么做。它会正常工作。 我只是坚持认为 `-` 和 `-=` 是不同的运算符，所以把它们放到不同行里。

把每个运算符放在单独的匹配中， 简化了正则表达式，代价是输入了额外的字符。 我喜欢这么做，但你可能不这么认为。你自己决定吧。

我也没有把 `=` 定义成一个运算符。我们等会会这么做，但我希望暂时先不这样做，这样就能给你考上一题了。

因为分开了 `-` 和 `-=` 的正则表达式，我不得不在定义 `-` 之后定义 `-=` ！

如果以相反的顺序定义，并在Potion文件里使用 `-=` ，Vim将匹配 `-` (当然，同时也高亮它)，剩下 `=` 等待匹配。这意味着在构建 `syntax match` 组时，每个组"消耗"的文本片段在之后不能被匹配到。

这讲得太笼统了，但我暂时并不打算纠结于细节。 总之，你应该在匹配较小的组之后匹配较大的组，因为在之后定义的组优先于在之前定义的组。

让我们继续并添加 `=` 作为运算符。现在请听题：

```
:::vim
syntax match potionOperator "\v\="
```

花一点时间想想你应该把它放在syntax文件的哪个位置。如果你需要提示，重读前几章。

练习

阅读 `:help syn-match` .

阅读 `:help syn-priority` .

在例子中，我们没有把 `:` 当作运算符。阅读Potion文档并审慎地决定是否把 `:` 当作一个运算符。 如果你决定这么做，把它加到syntax文件中。

同样考虑 `.` 和 `/` 。

增加一个高亮数字的语法类型分组 `potionNumber` 。链接它到高亮组 `Number` 。 不要忘了Potion支持 `2` ， `0xffaf` ， `123.23` ， `1e-2` 和 `1.9956e+2` 这几种形式。 记得在处理边际状态的花费的时间和这些边际状态出现的次数之间取得平衡。

更高级的语法高亮

我们甚至可以为Vim里面的语法高亮另开一本书了。

我们将在此讲解它最后的重要内容，然后继续讲别的东西。 如果你想要学到更多，去读 `:help syntax` 并阅读别人写的syntax文件。

高亮字符串

Potion，一如大多数编程语言，支持诸如 `"Hello,world!"` 的字符串字面量。 我们应该把这些高亮成字符串。为此我们将使用 `syntax region` 命令。 在你的Potion syntax文件中加入下面内容：

```
:::vim
syntax region potionString start=/\v"/ skip=/\v\\./ end=/\v"/
highlight link potionString String
```

关闭并重新打开你的 `factorial.pn` ，你将看到文件结尾的字符串被高亮了！

最后一行应该很熟了。如果你不懂，重读前两章。

第一行用一个"region"添加一个语法类型分组。 区域(Regions)有一个"start"模式和一个"end"模式来指定开头和结束的位置。 这里，一个Potion字符串从一个双引号开始，到另一个双引号结束。

`syntax region` 的"skip"参数允许我们处理转义字符串，比如 `"She said: \"Vimscript is tricky, but useful\"!"` 。

如果不提供 `skip` 参数，Vim将在 `Vimscript` 之前的 `"` 停止匹配字符串，这不是我们想要的！

简明扼要地说， `syntax region` 中的 `skip` 参数告诉Vim： "一旦你开始匹配这个区域，我希望你忽略 `skip` 匹配的内容，即使它会被当作区域结束的标志"。

花上几分钟去想透彻。如果遇到的是 `"foo \\" bar"` 会怎样？那会是正确的行为吗？ 那总是正确的行为吗？放下本书，花上几分钟来认真想一想！

练习

给单引号字符串加上语法高亮。

阅读 `:help syn-region` .

阅读 `:help syn-region` 将比阅读本章花费更多的时间。给自己倒杯饮料，这是你应得的！

基本折叠

如果从未在Vim里使用过代码折叠，你不知道你都错过了什么。 阅读 `:help usr_28` 并花费时间在日常工作中使用它。 一旦到了铭记于指的程度，你就可以继续本章了。

折叠类型

Vim支持六种不同的决定如何折叠你的文本的折叠类型。

Manual

你手动创建折叠并且折叠将被Vim储存在内存中。 当你关闭Vim时，它们也将一并烟消云散，而下次你编辑文件时将不得不重新创建。

在你把它跟一些自定义的创建折叠的映射结合起来时，这种方式会很方便。 在本书中，我们不会这么做，但当你想这么做的时候，它会帮上忙。

Marker

Vim基于特定的字符组合折叠你的代码。

这些字符通常放置于注释中(比如 `// {{{`)， 不过在有些语言里，你可以使用该语言自己的语法代替，比如javascript的 `{` 和 `}` 。

纯粹为了你的编辑器，用注释割裂你的代码看上去有点丑，但好处是你可以定制特定的折叠。 如果你想以特定的方式组织一个大文件，这个类型将是非常棒的选择。

Diff

在diff文件时使用该特定的折叠类型。我们不会讨论它，因为Vim会自动使用它。

Expr

这让你可以用自定义的Vimscript来决定折叠的位置。它是最为强大的方式，不过也需要最繁重的工作。 下一章我们将讲到它。

Indent

Vim使用你的代码的缩进来折叠。同样缩进等级的代码折叠到一块，空行则被折叠到周围的行一起去。

这是最便捷的方式，因为你的代码已经缩进过了；你仅仅需要启动它。 这将是我们用来折叠Potion代码的第一种方式。

Potion折叠

让我们再一次看一下Potion实例代码：

```
:::text
factorial = (n):
    total = 1
    n to 1 (i):
        total *= i.
    total.

10 times (i):
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.
```

函数体和循环体已经缩进好了。这意味着我们可以不怎么费力就能实现一些基本的缩进。

在我们开始之前，在 `total *= i` 上添加一个注释，这样我们就有一个供测试的多行内部块。你将在做练习的时候学到为什么我们需要这么做，但暂时先信任我。现在文件看上去就像这样：

```
:::text
factorial = (n):
    total = 1
    n to 1 (i):
        # Multiply the running total.
        total *= i.
    total.

10 times (i):
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.
```

在你的Potion插件的版本库下创建一个 `ftplugin` 文件夹，然后在里面创建一个 `potion` 文件夹。 最后，在 *`potion`* 文件夹里面创建一个 `folding.vim` 文件。

不要忘了每次Vim设置一个buffer的 `filetype` 为 `potion` 时，它都会执行这个文件中的代码。(因为它位于一个叫 `potion` 的文件夹)

将所有的折叠相关的代码放在同一个文件显然是一个好主意，它能帮我们维护我们的插件的繁多的功能。

在这个文件中加入下面一行：

```
:::vim
setlocal foldmethod=indent
```

关闭Vim，重新打开 `factoria.pn` 。用 `zR` ， `zM` 和 `za` 尝试折叠功能。

一行Vimscript代码就能带来一些有用的折叠！这真是太酷了！

你可能注意到 `factorial` 函数的内循环里面的那几行不能折叠，尽管它们缩进了。 为什么会这样？

事实上，在使用 `indent` 折叠时，Vim默认忽略以 `#` 字符开头的行。 这在编辑C文件时很有用(这时 `#` 表示一个预编译指令)，但在编辑其他文件时不怎么有意义。

让我们在 `ftplugin/potion/folding.vim` 中添加多一行来修复问题：

```
:::vim
setlocal foldmethod=indent
setlocal foldignore=
```

关闭并重新打开 `factorial.pn` ，现在内部块可以正常地折叠了。

练习

阅读 `:help foldmethod` .

阅读 `:help fold-manual` .

阅读 `:help fold-marker` 和 `:help foldmarker` .

阅读 `:help fold-indent` .

阅读 `:help fdl` 和 `:help foldlevelstart` .

阅读 `:help foldminlines` .

阅读 `:help foldignore` .

高级折叠

在上一章里我们用Vim的 `indent` 折叠方式，在Potion文件中增加了一些快捷而肮脏的折叠。

打开 `factorial.pn` 并用 `zM` 关闭所有的折叠。文件现在看起来就像这样：

```
:::text
factorial = (n):
+--  5 lines: total = 1

10 times (i):
+--  4 lines: i string print
```

展开第一个折叠，它看上去会是这样：

```
:::text
factorial = (n):
    total = 1
    n to 1 (i):
+---  2 lines: # Multiply the running total.
    total.

10 times (i):
+--  4 lines: i string print
```

这真不错，但我个人喜欢依照内容来折叠每个块的第一行。 在本章中我们将写下一些自定义的折叠代码，并在最后实现这样的效果：

```
:::text
factorial = (n):
    total = 1
+---  3 lines: n to 1 (i):
    total.

+--  5 lines: 10 times (i):
```

这将更为紧凑，而且(对我来说)更容易阅读。 如果你更喜欢 `indent` 也不是不行，不过最好学习本章来对Vim中实现折叠的代码的更深入的了解。

折叠原理

为了写好自定义的折叠，我们需要了解Vim对待("thinks")折叠的方式。简明扼要地讲解下规则：

- 文件中的每行代码都有一个"foldlevel"。它不是为零就是一个正整数。
- foldlevel为零的行不会被折叠。
- 有同等级的相邻行会被折叠到一起。
- 如果一个等级X的折叠被关闭了，任何在里面的、foldlevel不小于X的行都会一起被折叠，直到有一行的等级小于X。

通过一个例子，我们可以加深理解。打开一个Vim窗口然后粘贴下面的文本进去。

```
:::text
a
    b
    c
        d
        e
    f
g
```

执行下面的命令来设置 `indent` 折叠：

```
:::vim
:setlocal foldmethod=indent
```

花上一分钟玩一下折叠，观察它是怎么工作的。

现在执行下面的命令来看看第一行的foldlevel:

```
:::vim
:echom foldlevel(1)
```

Vim显示 `0` 。现在看看第二行的：

```
:::vim
:echom foldlevel(2)
```

Vim显示 `1` 。试一下第三行：

```
:::vim
:echom foldlevel(3)
```

Vim再次显示 `1` 。这意味着第2,3行都属于一个level1的折叠。

这是每一行的foldlevel:

```
:::text
a           0
    b       1
    c       1
        d   2
        e   2
    f       1
g           0
```

重读这一部分开头的几条规则。打开或关闭每个折叠，观察foldlevel，并确保你理解了为什么会这样折叠。

一旦你已经自信地认为你理解了每行的foldlevel是怎么影响折叠结构的，继续看下一部分。

首先：做一个规划

在我们埋头敲键盘之前，先为我们的折叠功能规划出几条大概的规则。

首先，同等缩进的行应该要折叠到一块。我们也希望上一行也一并折叠，达到这样的效果：

```
:::text
hello = (name):
    'Hello, ' print
    name print.
```

将折叠成这样：

```
:::text
+--  3 lines: hello = (name):
```

空行应该算入下一行，因此折叠底部的空行不会包括进去。这意味着类似这样的内容：

```
:::text
hello = (name):
    'Hello, ' print
    name print.

hello('Steve')
```

将折叠成这样：

```
:::text
+--  3 lines: hello = ():

hello('Steve')
```

而不是这样：

```
:::text
+--  4 lines: hello = ():
hello('Steve')
```

这当然是属于个人偏好的问题，但现在我们就这么定了。

开始

现在开始写我们的自定义折叠代码吧。 打开Vim，分出两个分割，一个是 `ftplugin/potion/folding.vim` ， 另一个是示例代码 `factorial.pn` 。

在上一章我们关闭并重新打开Vim来使得 `folding.vim` 生效，但其实还有更简单的方法。

不要忘记每当设置一个缓冲区的 `filetype` 为 `potion` 的时候，在 `ftplugin/potion/` 下的所有文件都会被执行。 这意味着仅需在 `factorial.pn` 的分割下执行 `:set ft=potion` ，Vim将重新加载折叠代码！

这比每次都关闭并重新打开文件要快多了。 唯一需要铭记的是，你得保存 `folding.vim` 到硬盘上，否则未保存的改变不会起作用。

Expr折叠

为了获取折叠上的无限自由，我们将使用Vim的 `expr` 折叠。

我们可以继续并从 `folding.vim` 移除 `foldignore` ，因为它只在使用 `indent` 的时候生效。 我们也打算让Vim使用 `expr` 折叠，所以把 `folding.vim` 改成这样：

```
:::vim
setlocal foldmethod=expr
setlocal foldexpr=GetPotionFold(v:lnum)

function! GetPotionFold(lnum)
    return '0'
endfunction
```

第一行只是告诉Vim使用 `expr` 折叠。

第二行定义了Vim用来计算每一行的foldlevel的表达式。 当Vim执行某个表达式，它会设置 `v:lnum` 为它需要的对应行的行号。 我们的表达式将把这个数字作为自定义函数的参数。

最后我们定义一个对任意行均返回 `0` 的占位(dummy)函数。 注意它返回的是一个字符串而不是一个整数。等会我们就知道为什么这么做。

继续并重新加载折叠代码(保存 `folding.vim` 并对 `factorial.pn` 执行 `:set ft=potion`)。 我们的函数对任意行均返回 `0` ,所以Vim将不会进行任何折叠。

空行

让我们先解决空行的特殊情况。修改 `GetPotionFold` 函数成这样：

```
:::vim
function! GetPotionFold(lnum)
    if getline(a:lnum) =~? '\v^\s*$'
        return '-1'
    endif

    return '0'
endfunction
```

我们增加了一个 `if` 语句来处理空行。它是怎么起效的？

首先，我们使用 `getline(a:lnum)` 来以字符串形式获取当前行的内容。

我们把结果跟正则表达式 `\v^\s*$` 比较。记得 `\v` 表示"very magic"(我的意思是，正常的)模式。 这个正则表达式将匹配"行的开头，任何空白字符，行的结尾"。

比较是用大小写不敏感比较符 `=~?` 完成的。 技术上我们不用担心大小写，毕竟我们只匹配空白，但是我偏好在比较字符串时使用更清晰的方式。 如果你喜欢，可以使用 `=~` 代替。

如果需要唤起Vim中的正则表达式的回忆，你应该回头重读"基本正则表达式"和"Grep Operator"这两部分。

如果当前行包括一些非空白字符，它将不会匹配，我们将如前返回 `0` 。

如果当前行匹配正则表达式(i.e. 比如它是空的或者只有空格)，就返回字符串 `'-1'` 。

之前我说过一行的foldlevel可以为0或者正整数，所以这会发生什么？

特殊折叠

你自定义的表达式可以直接返回一个foldlevel，或者返回一个"特殊字符串"来告诉Vim如何折叠这一行。

`'-1'` 正是其中一种特殊字符串。它告知Vim，这一行的foldlevel为"undefined"。 Vim将把它理解为"该行的foldlevel等于其上一行或下一行的较小的那个foldlevel"。

这不是我们计划中的最终结果，但我们可以看到，它已经足够接近了，而且必将达到我们的目标。

Vim可以把undefined的行串在一起，所以假设你有三个undefined的行和接下来的一个level1的行， 它将设置最后一行为1,接着是倒数第二行为1,然后是第一行为1。

在写自定义的折叠代码时，你经常会发现有几种行你可以容易地设置好它们的foldlevel。 然后你就可以使用 `'-1'` (或我们等会会看到的其他特殊foldlevel)来"瀑布般地"设置好剩余的行的foldlevel。

如果你重新加载了 `factorial.pn` 的折叠代码，Vim依然不会折叠任何行。 这是因为所有的行的foldlevel要不是为0,就是为"undefined"。 等级为0的行将影响undefined的行，最终导致所有的行的foldlevel都是 `0` 。

缩进等级辅助函数

为了处理非空行，我们需要知道它们的缩进等级，所以让我们来创建一个辅助函数替我们计算它。 在 `GetPotionFold` 之上加上下面的函数：

```
:::vim
function! IndentLevel(lnum)
    return indent(a:lnum) / &shiftwidth
endfunction
```

重新加载折叠代码。在 `factorial.pn` 缓冲区执行下面的命令来测试你的函数：

```
:::vim
:echom IndentLevel(1)
```

Vim显示 `0` ,因为第一行没有缩进。现在在第二行试试看：

```
:::vim
:echom IndentLevel(2)
```

这次Vim显示 `1` 。第二行开头有四个空格，而 `shiftwidth` 设置为4,所以4除以4得1。

我们用它除以缓冲区的 `shiftwidth` 来得到缩进等级。

为什么我们使用 `&shiftwidth` 而不是直接除以4？如果有人偏好使用2个空格缩进他们的Potion代码，除以4将导致不正确的结果。使用 `shiftwidth` 可以允许任何缩进的空格数。

再来一个辅助函数

下一步的方向尚未明朗。让我们停下来想想为了确定折叠非空行，还需要什么信息。

我们需要知道每一行的缩进等级。我们已经通过 `IndentLevel` 函数得到了，所以这个条件已经满足了。

我们也需要知道下一个非空行的缩进等级，因为我们希望折叠段头行到对应的缩进段中去。

让我们写一个辅助函数来得到给定行的下一个非空行的foldlevel。在 `IndentLevel` 上面加入下面的函数：

```
:::vim
function! NextNonBlankLine(lnum)
    let numlines = line('$')
    let current = a:lnum + 1

    while current <= numlines
        if getline(current) =~? '\v\S'
            return current
        endif

        let current += 1
    endwhile

    return -2
endfunction
```

这个函数有点长，不过很简单。让我们逐个部分分析它。

首先我们用 `line('$')` 得到文件的总行数。查查文档来了解 `line()` 。

接着我们设变量 `current` 为下一行的行号。

然后我们开始一个会遍历文件中每一行的循环。

如果某一行匹配正则表达式 `\v\S` ，表示匹配"有一个非空白字符"，它就是非空行，所以返回它的行号。

如果某一行不匹配，我们就循环到下一行。

如果循环到达文件尾行而没有任何返回，这就说明当前行之后没有非空行！我们返回 `-2` 来指明这种情况。 `-2` 不是一个有效的行号，所以用来简单地表示"抱歉，没有有效的结果"。

我们可以返回 `-1` ，因为它也是一个无效的行号。 我甚至可以选择 `0` ,因为Vim中的行号从 `1` 开始！ 所以为何我选择 `-2` 这个看上去奇怪的选项？

我选择 `-2` 是因为我们正处理着折叠代码，而 `'-1'` (和 `'0'`)是特殊的Vim foldlevel字符串。

当眼睛正扫过代码时，看到 `-1` ，脑子里会立刻浮现起"undefined foldlevel"。 这对于 `0` 也差不多。 我在这里选择 `-2` ，就是为了突出它不是foldlevel，而是表示一个"错误"。

如果你觉得这不可理喻，你可以安心地替换 `-2` 为 `-1` 或 `0` 。 这只是代码风格问题。

完成折叠函数

本章已经显得比较冗长了，所以现在把折叠函数包装起来(wrap up)吧。把 `GetPotionFold` 修改成这样：

```
:::vim
function! GetPotionFold(lnum)
    if getline(a:lnum) =~? '\v^\s*$'
        return '-1'
    endif

    let this_indent = IndentLevel(a:lnum)
    let next_indent = IndentLevel(NextNonBlankLine(a:lnum))

    if next_indent == this_indent
        return this_indent
    elseif next_indent < this_indent
        return this_indent
    elseif next_indent > this_indent
        return '>' . next_indent
    endif
endfunction
```

这里的新代码真多！让我们分开一步步来看。

空行

首先我们检查空行。这里没有改动。

如果不是空行，我们就准备好处理非空行的情况了。

获取缩进等级

接下来我们使用两个辅助函数来获取当前行和下一个非空行的折叠等级。

你可能会疑惑万一 `NextNonBlankLine` 返回错误码 `-2` 该怎么办。 如果这发生了， `indent(-2)` 还会继续工作。对一个不存在的行号执行 `indent()` 将返回 `-1` 。 你可以试试 `:echom indent(-2)` 看看。

`-1` 除以任意大于1的 `shiftwidth` 将返回 `0` 。 这好像有问题，不过它实际上不会有。现在暂时不用纠结于此。

同级缩进

既然我们已经得到了当前行和下一非空行的缩进等级，我们可以比较它们并决定如何折叠当前行。

这里又是一个 `if` 语句：

```
:::vim
if next_indent == this_indent
    return this_indent
elseif next_indent < this_indent
    return this_indent
elseif next_indent > this_indent
    return '>' . next_indent
endif
```

首先我们检查这两行是否有同样的缩进等级。如果相等，我们就直接把缩进等级当作foldlevel返回！

举个例子：

```
:::text
a
b
    c
    d
e
```

假设我们正处理包含 `c` 的那一行，它的缩进等级为1。 下一个非空行("d")的缩进等级也是一样的，所以返回 `1` 作为foldlevel。

假设我们正处理"a"，它的缩进等级为0。这跟下一非空行("b")的等级是一样的，所以返回 `0` 作为foldlevel。

在这个简单的示例中，可以分出两个foldlevel。

```
:::text
a       0
b       ?
    c   1
    d   ?
e       ?
```

纯粹出于运气，这种情况也处理了在最后一行对特殊的"error"情况。记得我们说过，如果我们的辅助函数返回 `-2` , `next_indent` 将会是 `0` 。

在这个例子中，行"e"的缩进等级为 `0` ，而 `next_indent` 也被设为 `0` ，所以匹配这种情况并返回 `0` 。现在foldlevels是这样：

```
:::text
a       0
b       ?
    c   1
    d   ?
e       0
```

更低的缩进等级

我们再来看看那个 `if` 语句：

```
:::vim
if next_indent == this_indent
    return this_indent
elseif next_indent < this_indent
    return this_indent
elseif next_indent > this_indent
    return '>' . next_indent
endif
```

`if` 的第二部分检查下一行的缩进等级是否比当前行小。就像是例子中行"d"的情况。

如果符合，将再一次返回当前行的缩进等级。

现在我们的例子看起来像这样：

```
:::text
a       0
b       ?
    c   1
    d   1
e       0
```

当然，你可以用 `||` 把两种情况连接起来，但是我偏好分开来写以显得更清晰。 你的想法可能不同。这只是风格问题。

又一次，纯粹出于运气，这种情况处理了其他来自辅助函数的"error"状态。设想我们有一个文件像这样：

```
:::text
a
    b
    c
```

第一种情况处理行"b"：

```
:::text
a       ?
    b   1
    c   ?
```

行"c"为最后一行，有着缩进等级1。由于我们的辅助函数， `next_indent` 将设为 `0` 。 这匹配 `if` 语句的第二部分，所以foldlevel设为当前缩进等级，也即是 `1` 。

```
:::text
a       ?
    b   1
    c   1
```

结果如我们所愿，"b"和"c"折叠到一块去了。

更高的缩进等级

现在还剩下最后一个 `if` 语句：

```
:::vim
if next_indent == this_indent
    return this_indent
elseif next_indent < this_indent
    return this_indent
elseif next_indent > this_indent
    return '>' . next_indent
endif
```

而我们的例子现在是：

```
:::text
a       0
b       ?
    c   1
    d   1
e       0
```

只剩下行"b"我们还不知道它的foldlevel，因为：

- "b"的缩进等级为 `0` 。
- "c"的缩进等级为 `1` 。
- 1既不等于0,又不小于0。

最后一种情况检查下一行的缩进等级是否大于当前行。

这种情况下Vim的 `indent` 折叠并不理想，也是为什么我们一开始打算写自定义的折叠代码的原因！

最后的情况表示，当下一行的缩进比当前行多，它将返回一个以 `>` 开头和下一行的缩进等级构成的字符串。 这是什么意思呢？

从折叠表达式中返回的，类似 `>1` 的字符串表示Vim的特殊foldlevel中的一种。 它告诉Vim当前行需要展开一个给定level的折叠。

在这个简单的例子中，我们可以简单返回表示缩进等级的数字，但我们很快将看到为什么要这么做。

这种情况下"b"将展开level1的折叠，使我们的例子变成这样：

```
:::text
a       0
b       >1
    c   1
    d   1
e       0
```

这就是我们想要的！万岁！

复习

如果你一步步做到了这里，你应该为自己感到骄傲。即使像这样的简单折叠代码，也会是令人绞尽脑汁的。

在我们结束之前，让我们重温最初的 `factorial.pn` 代码，看看我们的折叠表达式是怎么处理每一行的foldlevel的。

重新把 `factorial.pn` 代码列在这里：

```
:::text
factorial = (n):
    total = 1
    n to 1 (i):
        # Multiply the running total.
        total *= i.
    total.

10 times (i):
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.
```

首先，所有的空行的foldlevel都将设为undefined：

```
:::text
factorial = (n):
    total = 1
    n to 1 (i):
        # Multiply the running total.
        total *= i.
    total.
                                         undefined
10 times (i):
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.
```

所有折叠等级跟下一行的相等的行，它们的foldlevel等于折叠等级：

```
:::text
factorial = (n):
    total = 1                            1
    n to 1 (i):
        # Multiply the running total.    2
        total *= i.
    total.
                                         undefined
10 times (i):
    i string print                       1
    '! is: ' print                       1
    factorial (i) string print           1
    "\n" print.
```

在下一行的缩进比当前行更少的情况下，也是同样的处理：

```
:::text
factorial = (n):
    total = 1                         1
    n to 1 (i):
        # Multiply the running total.    2
        total *= i.                   2
    total.                            1
                                      undefined
10 times (i):
    i string print                    1
    '! is: ' print                    1
    factorial (i) string print        1
    "\n" print.                       1
```

最后的情况是下一行的缩进比当前行更多。如果这样，那就设当前行的折叠等级为展开下一行的折叠：

```
:::text
factorial = (n):                      >1
    total = 1                         1
    n to 1 (i):                       >2
        # Multiply the running total.    2
        total *= i.                   2
    total.                            1
                                      undefined
10 times (i):                         >1
    i string print                    1
    '! is: ' print                    1
    factorial (i) string print        1
    "\n" print.                       1
```

现在我们已经得到了文件中每一行的foldlevel。剩下的就是由Vim来解决未定义(undefined)的行。

不久前我说过undefined的行将选择相邻行中较小的那个foldlevel。

Vim手册是这么讲的，但不是十分地确切。 如果真是这样的，我们的文件中的空行的foldlevel为1，因为它相邻两行的foldlevel都为1。

事实上，空行的foldlevel将被设定成0！

这就是为什么我们不直接设置 `10 times(i):` 的foldlevel为1。我们告诉Vim该行展开一个level1的折叠。 Vim能够意识到这意味着undefined的行应该设置成 `0` 而不是 `1` 。

这样做背后的理由也许深埋在Vim的源码里。 通常Vim在处理undefined行时，对待特殊的foldlevel的行为都是很聪明的，所以你总能如愿以偿。

一旦Vim处理完undefined行，它会得到一个对每一行的折叠情况的完整描述，看上去像这样：

```
:::text
factorial = (n):                         1
    total = 1                            1
    n to 1 (i):                          2
        # Multiply the running total.    2
        total *= i.                      2
    total.                               1
                                         0
10 times (i):                            1
    i string print                       1
    '! is: ' print                       1
    factorial (i) string print           1
    "\n" print.                          1
```

这就是了，我们完成啦！重新加载折叠代码，在 `factorial.pn` 中玩玩我们神奇的折叠功能吧！

练习

阅读 `:help foldexpr` .

阅读 `:help fold-expr` 。注意你的表达式可以返回的所有特殊字符串。

阅读 `:help getline` 。

阅读 `:help indent()` 。

阅读 `:help line()` 。

想想为什么我们用 `.` 连接 `>` 和我们折叠函数给出的数字。如果我们使用的是 `+` 会怎样？

我们在全局空间中定义了辅助函数，但这不是好的做法。把它改到脚本本地的命名空间中。

放下本书，出去玩一下，让你的大脑从本章中清醒清醒。

段移动原理

如果你未曾用过Vim的段移动命令 (`[[` , `]]` , `[]` and `][`)，现在花上几秒读读它们的帮助文档。 也顺便读读 `:help section` 。

还是不懂？这不是什么问题，我第一次读这些的时候也是这样。 在写代码之前，我们先岔开来学习这些移动是怎么工作的，然后在下一章我们将使得我们的Potion插件支持它们。

Nroff文件

四个"段移动"命令正如其字面上的含义，可以用来在文件的"段"之间移动。

这些命令默认为[nroff文件][]而设计。 Nroff类似于LaTex或Markdown -- 它是用来写标记文本的(最终会生成UNIX man页面)。

Nroff文件使用一组"macro"来定义"段头"。 比如，这里有个来自于 `awk` man页面的例子：

```
:::nroff
.SH NAME                                                            ***
awk \- pattern-directed scanning and processing language
.SH SYNOPSIS                                                        ***
.B awk
[
.BI \-F
.I fs
]
[
.BI \-v
.I var=value
]
[
.I 'prog'
|
.BI \-f
.I progfile
]
[
.I file ...
]
.SH DESCRIPTION                                                     ***
.I Awk
scans each input
.I file
for lines that match ...
```

以 `.SH` 开头的行就是段头。我用 `***` 把它们标记出来。 四个段移动命令将在段头行之间移动你的光标。

Vim以 `.` 和nroff的段头符开始的任何行当做一个段头，即使你编辑的不是*nroff*文件！

你可以改变 `sections` 设置来改变段头符，但Vim依旧需要在行开头有一个点，而且段头符必须是成对的字符， 所以这样改对Potion文件不会有足够的灵活性。

括号

段移动命令也查看另一样东西：一个打开或关闭的大括号(`{` 或 `}`)作为行的第一个字符。

`[[` 和 `]]` 查看开括号，而 `[]` 和 `][` 查看闭括号。

这额外的"行为"使得你可以在C风格语言的段之间轻松移动。 然而，这些规则也依旧没有顾及你正在编辑的文件类型！

加入下面内容到一个缓冲区里：

```
:::text
Test           A B
Test

.SH Hello      A B

Test

{              A
Test
}                B

Test

.H World       A B

Test
Test           A B
```

现在执行 `:set filetype=basic` 来告诉Vim这是一个BASIC文件，并尝试段移动命令。

`[[` 和 `]]` 命令将在标记为 `A` 的行之间移动，而 `[]` 和 `][` 将在标记为 `B` 的行之间移动。

这告诉我们，Vim总是用同样的两条规则来处理段移动，即使没有一条是起作用的(比如在BASIC中的情况)！

练习

再次阅读 `:help section` ，现在你应该可以理解段移动了。

也顺便读读 `:help sections` 吧。

Potion段移动

既然知道了段移动的工作原理，让我们重新映射这些命令来使得它们对于Potion文件起作用。

首先我们要决定Potion文件中"段"的意义。 有两对段移动命令，所以我们可以总结出两套组合，我们的用户可以选择自己喜欢的一个。

让我们使用下面两个组合来决定哪里是Potion中的段：

1. 任何在空行之后的，第一个字符为非空字符的行，以及文件首行。
2. 任何第一个字符为非空字符，包括一个等于号，并以冒号结尾的行。

稍微拓展我们的 `factorial.pn` 例子，这就是那些规则当作段头的地方：

```
:::text
# factorial.pn                            1
# Print some factorials, just for fun.

factorial = (n):                          1 2
    total = 1

    n to 1 (i):
        total *= i.

    total.

print_line = ():                          1 2
    "-=-=-=-=-=-=-=-\n" print.

print_factorial = (i):                    1 2
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.

"Here are some factorials:\n\n" print     1

print_line ()                             1
10 times (i):
    print_factorial (i).
print_line ()
```

我们的第一个定义更加自由。它定义一个段为一个"顶级的文本块"。

第二个定义则严格一点。它定义一个段为一个函数定义。

自定义映射

在你的插件的repo中创建 `ftplugin/potion/sections.vim` 。 这将是我们放置段移动代码的地方。记得一旦一个缓冲区的 `filetype` 设置为 `potion` ，这里的代码就会执行。

我们将重新映射全部四个段移动命令，所以继续并创建一个骨架：

```
:::vim
noremap <script> <buffer> <silent> [[ <nop>
noremap <script> <buffer> <silent> ]] <nop>

noremap <script> <buffer> <silent> [] <nop>
noremap <script> <buffer> <silent> ][ <nop>
```

Notice that we use `noremap` commands instead of `nnoremap` , because we want these to work in operator-pending mode too. That way you'll be able to do things like `d]]` to "delete from here to the next section". 注意我们使用 `noremap` 而不是 `nnoremap` ，因为我们想要这些命令也能在operator-pending模式下工作。 这样你就能使用 `d]]` 命令来删除从这到下一段之间的内容。

我们设置映射生效于buffer-local，所以它们只对Potion文件起作用，不会替换全局选项。

我们也设置了silent，因为用户不应关心我们实现段移动的细节。

使用一个函数

每个命令中实现段移动的代码会是非常相似的，所以让我们把它抽象出供映射调用的一个函数。

你将在那些创建了一些相似的映射的Vim插件中频繁看到这种策略。 比起把所有的功能堆砌于各个映射中，这样做不仅更易读，而且更易维护。

在 `sections.vim` 文件中加上下面内容：

```
:::vim
function! s:NextSection(type, backwards)
endfunction

noremap <script> <buffer> <silent> ]]
        \ :call <SID>NextSection(1, 0)<cr>

noremap <script> <buffer> <silent> [[
        \ :call <SID>NextSection(1, 1)<cr>

noremap <script> <buffer> <silent> ][
        \ :call <SID>NextSection(2, 0)<cr>

noremap <script> <buffer> <silent> []
        \ :call <SID>NextSection(2, 1)<cr>
```

这里我用到了Vimscript的断行特性，因为我不想看到又长又臭的代码。 注意反斜杠是放在第二行前面进行转义的。阅读 `:help line-continuation` 以了解更多。

注意我们使用 `<SID>` 和一个脚本本地命名空间内定义的函数来避免污染全局空间。

每个映射简单地以适当参数调用 `NextSection` 实现对应的移动。 现在我们可以开始实现 `NextSection` 了。

基本移动

让我们考虑下我们的函数需要做什么。 我们想要移动光标到下一段，而移动光标，有一个简单的办法就是利用 `/` 和 `?` 命令。

编辑 `NextSection` 成这样：

```
:::vim
function! s:NextSection(type, backwards)
    if a:backwards
        let dir = '?'
    else
        let dir = '/'
    endif

    execute 'silent normal! ' . dir . 'foo' . "\r"
endfunction
```

现在这个函数使用我们之前见过的 `execute normal!` 来执行 `/foo` 或 `?foo` ，取决于 `backwards` 的值。 这将是个好的开始。

继续前进，我们显然需要搜索 `foo` 以外的东西，是什么则取决于用的是段头的第一个还是第二个定义。

把 `NextSection` 改成这样：

```
:::vim
function! s:NextSection(type, backwards)
    if a:type == 1
        let pattern = 'one'
    elseif a:type == 2
        let pattern = 'two'
    endif

    if a:backwards
        let dir = '?'
    else
        let dir = '/'
    endif

    execute 'silent normal! ' . dir . pattern . "\r"
endfunction
```

现在只需要补上匹配的模式了(pattern)，让我们继续完成它吧。

顶级文本段

用下面一行替换掉第一个 `let pattern = '...'` ：

```
:::vim
let pattern = '\v(\n\n^\S|%^)'
```

如果不理解这个正则表达式是干什么的，请回忆我们正在实现的"段"的定义。

> 任何在空行之后的，第一个字符为非空字符的行，以及文件首行。

开头的 `\v` 强制切换为"very magic"模式，一如之前的几次。

剩下的正则表达式由两个选项组成。第一个， `\n\n^\S` ，搜索"两个换行符，接着之后是一个非空字符"。 这正好是我们的定义中的第一种情况。

另一个是 `%^` ，在Vim中，这是一个代表文件开头的特殊正则符号。

我们现在已经到了尝试前两个映射的时机了。 保存 `ftplugin/potion/sections.vim` 并在你的Potion例子缓冲区中执行 `:set filetype=potion` 。 `[[` 和 `]]` 命令应该可以工作，但会显得古怪。

搜索标记

你大概注意到了，在段之间移动时光标会位于真正想要移动到的地方上方的空行。 在继续阅读之前，先想想为什么会这样。

问题在于我们使用 `/` (或 `?`)进行搜索，而在默认情况下Vim会把光标移动到匹配开始处。 举个例子，当你执行 `/foo` 光标会位于 `foo` 中的 `f` 。

为了让Vim把光标移动到匹配结束处而不是开始处，我们可以使用搜索标记(search flag)。 试试在Potion文件中这么搜索：

```
:::vim
/factorial/e
```

Vim将找到 `factorial` 并带你到那。按下几次 `n` 来在匹配处之间移动。 `e` 标记将使得Vim把光标移动到到匹配结束处而不是开始处。在另一个方向也试试：

```
:::vim
?factorial?e
```

让我们来修改我们的函数，用搜索标记来放置光标到匹配的段头的另一端。

```
:::vim
function! s:NextSection(type, backwards)
    if a:type == 1
        let pattern = '\v(\n\n^\S|%^)'
        let flags = 'e'
    elseif a:type == 2
        let pattern = 'two'
        let flags = ''
    endif

    if a:backwards
        let dir = '?'
    else
        let dir = '/'
    endif

    execute 'silent normal! ' . dir . pattern . dir . flags . "\r"
endfunction
```

我们这里改动了两处。首先，我们依照段移动的类型设置 `flags` 变量的值。 现在我们仅需处理第一种情况，所以设置了标记 `e` 。

其次，我们在搜索字符串中连接 `dir` 和 `flags` 。这将依照我们搜索的方向加入 `?e` 或 `/e` 。

保存文件，切换回Potion示例文件，并执行 `:set ft=potion` 来让改动生效。 现在尝试 `[[` 和 `]]` 来看看我们的成果吧！

函数定义

是时候处理我们对"段"的第二个定义了，幸运的是这个比起第一个简单多了。 重新说一下我们需要实现的定义：

> 任何第一个字符为非空字符，包括一个等于号，并以冒号结尾的行。

我们可以使用一个简单的正则表达式来查找这样的行。 修改函数中第二个 `let pattern = '...'` 成这样：

```
:::vim
let pattern = '\v^\S.*\=.*:$'
```

这个正则表达式比上一个没那么吓人多了。我把指出它是怎么工作的任务作为你的练习 -- 它只是我们的定义的一个直白的翻译。

保存文件，在 `factorial.pn` 处执行 `:set filetype=potion` ，然后试试新的 `][` 和 `[]` 映射。它们应该能如期工作。

在这里我们不需要搜索标记，因为默认的移动到匹配处开头正是我们想要的。

可视模式

我们的段移动命令在normal模式下一切正常，但要让它们也能在visual模式下工作，我们还需要增加一些东西。 首先，把函数改成这样：

```
:::vim
function! s:NextSection(type, backwards, visual)
    if a:visual
        normal! gv
    endif

    if a:type == 1
        let pattern = '\v(\n\n^\S|%^)'
        let flags = 'e'
    elseif a:type == 2
        let pattern = '\v^\S.*\=.*:$'
        let flags = ''
    endif

    if a:backwards
        let dir = '?'
    else
        let dir = '/'
    endif

    execute 'silent normal! ' . dir . pattern . dir . flags . "\r"
endfunction
```

Two things have changed. First, the function takes an extra argument so it knows whether it's being called from visual mode or not. Second, if it's called from visual mode we run `gv` to restore the visual selection. 两个地方改变了。首先，函数接受的参数多了一个，这样它能知道自己是否是在visual模式下调用的。 其次，如果它是在visual模式下调用的，我们执行 `gv` 来恢复可视选择区域。

为什么我们要这么做？来，让我展示给你看。 在visual模式下随意选择一些文本并执行下面命令：

```
:::vim
:echom "hello"
```

Vim将显示 `hello` ，但可视模式下选择的范围也随之清空！

当用 `:` 执行一个ex模式下的命令，可视选择的范围总会被清空。 `gv` 命令重新选择之前的可视选择范围，相当于撤销了清空。 这是个有用的命令，你会在日常工作中因此受益的。

现在我们需要更新前面的映射，传递 `0` 给新的 `visual` 参数：

```
:::vim
noremap <script> <buffer> <silent> ]]
        \ :call <SID>NextSection(1, 0, 0)<cr>

noremap <script> <buffer> <silent> [[
        \ :call <SID>NextSection(1, 1, 0)<cr>

noremap <script> <buffer> <silent> ][
        \ :call <SID>NextSection(2, 0, 0)<cr>

noremap <script> <buffer> <silent> []
        \ :call <SID>NextSection(2, 1, 0)<cr>
```

这里没什么是过于复杂的。现在让我们加上visual模式映射，作为最后一块拼图。

```
:::vim
vnoremap <script> <buffer> <silent> ]]
        \ :<c-u>call <SID>NextSection(1, 0, 1)<cr>

vnoremap <script> <buffer> <silent> [[
        \ :<c-u>call <SID>NextSection(1, 1, 1)<cr>

vnoremap <script> <buffer> <silent> ][
        \ :<c-u>call <SID>NextSection(2, 0, 1)<cr>

vnoremap <script> <buffer> <silent> []
        \ :<c-u>call <SID>NextSection(2, 1, 1)<cr>
```

这些映射都设置 `visual` 参数的值为 `1` ，来告诉Vim在移动之前重新选择上一次的选择范围。这里也用到了我们在Grep Operator那几章学到的 `<c-u>` 技巧。

保存文件，在Potion文件中 `set ft=potion` ，大功告成！尝试一下你的新映射吧。像 `v]]` 和 `d[]` 这样的命令现在应该可以正常地工作了。

我们得到了什么？

这是冗长的一章，尽管我们只实现了一些看上去简单的功能，但是你学到了(并充分地练习了)下列有用的知识：

- 使用 `noremap` 而不是 `nnoremap` 来创建可以作为移动和动作使用的命令。
- 在创建相关联的映射时，使用一个单一的接受多个参数的函数来简化你的工作。
- 逐渐增强一个Vimscript函数的能力。
- 动态地组建一个`execute 'normal! ...'字符串。
- 结合正则表达式，使用简单的搜索来实现移动。
- 使用特殊的正则符号，比如 `%^` (文件开头)。

- 使用搜索标记来改变搜索的行为。
- 实现不会改变可视选择范围的visual模式映射

坚持下并完成练习(不过是阅读一些文档)，然后赏自己一些冰激凌。你值得拥有！

练习

阅读 `:help search()` 。这是一个值得了解的函数，不过你也可以使用跟 `/` 和 `?` 列在一起的标记。

阅读 `:help ordinary-atom` 来认识能在搜索模式(pattern)中用到的更多有趣的东西。

外部命令

Vim遵循UNIX哲学"做一件事，做好它"。 与其试图集成你可能想要的功能到编辑器自身，更好的办法是在适当时使用Vim来调用外部命令。

让我们在插件中添加一些跟Potion编译器交互的命令，来浅尝在Vim里面调用外部命令的方法。

编译

首先我们将加入一个命令来编译和执行当前Potion文件。 有很多方法可以实现这一点，不过我们暂且用外部命令简单地实现。

在你的插件的repo中创建 `potion/ftplugin/potion/running.vim` 文件。 这将是我们创建编译和执行Potion文件的映射的地方。

```
:::vim
if !exists("g:potion_command")
    let g:potion_command = "potion"
endif

function! PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction

nnoremap <buffer> <localleader>r :call PotionCompileAndRunFile()<cr>
```

第一部分以全局变量的形式储存着用来执行Potion代码的命令，如果还没有设置过的话。 我们之前见过类似的检查了。

如果 `potion` 不在用户的 `$PATH` 内，这将允许用户覆盖掉它， 比如在 `~/.vimrc` 添加类似 `let g:potion_command = "/Users/sjl/src/potion/potion"` 的一行。

最后一行添加了一个buffer-local的映射来调用我们前面定义的函数。 不要忘了，由于这个文件位于 `ftdetect/potion` 文件夹，每次一个文件的 `filetype` 设置成 `potion` ，它都会被执行。

真正实现了功能的地方在 `PotionCompileAndRunFile()` 。 保存文件，打开 `factorial.pn` 并按下 `<localleader>r` 来执行这个映射，看看发生了什么。

如果 `potion` 位于你的 `$PATH` 下，代码会被执行，你应该在终端看到输出(或者在窗口底部，如果你用的是GUI vim)。 如果你看到了没有找到 `potion` 命令的错误，你需要像上面提到那样在 `~/.vimrc` 内设置 `g:potion_command` 。

让我们了解一下 `PotionCompileAndRunFile()` 的工作原理。

Bang!

`:!` 命令(念作"bang")会执行外部命令并在屏幕上显示它们的输出。尝试执行下面的命令：

```
:::vim
:!ls
```

Vim将输出 `ls` 命令的结果，同时还有"请按 ENTER 或其它命令继续"的提示。

当这样执行时，Vim不会传递任何输入给外部命令。为了验证，执行：

```
:::vim
:!cat
```

打入一些行，然后你将看到 `cat` 命令把它们都吐回来了，就像你是在Vim之外执行 `cat` 。按下Ctrl-D来结束。

想要执行一个外部命令并避免 `请按 ENTER 或其它命令继续` 的提示，使用 `:silent !` 。执行下面的命令：

```
:::vim
:silent !echo Hello, world.
```

如果在GUI Vim比如MacVim或gVim下执行，你将不会看到 `Hello,world.` 的输出。

如果你在终端下执行，你看到的结果取决于你的配置。 一旦执行了一个 `:silent !` ，你可能需要执行 `:redraw!` 来重新刷新屏幕。

注意这个命令是 `:silent !` 而不是 `:silent!` (看到空格了吗？)！ 这是两个不一样的命令，我们想要的是前者！Vimscript奇妙吧？

让我们回到 `PotionCompileAndRun()` 上来：

```
:::vim
function! PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction
```

首先我们执行一个 `silent !clear` 命令，来清空屏幕输出并避免产生提示。 这将确保我们仅仅看到本次命令的输出，如果一再执行同样的命令，你会觉得有用的。

在下一行我们使用老朋友 `execute` 来动态创建一个命令。建立的命令看上去类似于：

```
:::text
!potion factorial.pn
```

注意这里没有 `silent` ，所以用户将看到命令输出，并不得不按下enter来返回Vim。 这就是我们想要的，所以就这样设置好了。

显示字节码

Potion编译器有一个显示由它生成的字节码的选项。如果你正试图在非常低级的层次下debug，这将帮上忙。 在shell里执行下面的命令：

```
:::text
potion -c -V factorial.pn
```

你将看到一大堆像这样的输出：

```
:::text
-- parsed --
code ...
-- compiled --
; function definition: 0x109d6e9c8 ; 108 bytes
; () 3 registers
.local factorial ; 0
.local print_line ; 1
.local print_factorial ; 2
...
[ 2] move     1 0
[ 3] loadk    0 0   ; string
[ 4] bind     0 1
[ 5] loadpn   2 0   ; nil
[ 6] call     0 2
...
```

让我们添加一个使用户可以在新的Vim分割下，看到当前Potion代码生成的字节码的映射， 这样他们能更方便地浏览并测试输出。

首先，在 `ftplugin/potion/running.vim` 底部添加下面一行：

```
:::vim
nnoremap <buffer> <localleader>b :call PotionShowBytecode()<cr>
```

这里没有什么特别的 -- 只是一个简单的映射。现在先描划出函数的大概框架：

```
:::vim
function! PotionShowBytecode()
    " Get the bytecode.

    " Open a new split and set it up.

    " Insert the bytecode.

endfunction
```

既然已经建立起一个框架，让我们把它变成现实吧。

system()

有许多不同的方法可以实现这一点，所以我选择相对便捷的一个。

执行下面的命令：

```
:::vim
:echom system("ls")
```

你应该在屏幕的底部看到 `ls` 命令的输出。如果执行 `:message` ，你也能看到它们。 Vim函数 `system()` 接受一个字符串命令作为参数并以字符串形式返回那个命令的输出。

你可以把另一个字符串作为参数传递给 `system()` 。执行下面命令：

```
:::vim
:echom system("wc -c", "abcdefg")
```

Vim将显示 `7` (以及一些别的)。 如果你像这样传递第二个参数，Vim将写入它到临时文件中并通过管道作为标准输入输入到命令里。 目前我们不需要这个特性，不过它值得了解。

回到我们的函数。编辑 `PotionShowBytecode()` 来填充框架的第一部分：

```
:::vim
function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))
    echom bytecode

    " Open a new split and set it up.

    " Insert the bytecode.

endfunction
```

保存文件，在 `factorial.pn` 处执行 `:set ft=potion` 重新加载，并使用 `<lovalleader>b` 尝试一下。 Vim会在屏幕的底部显示字节码。一旦看到它成功执行了，你可以移除 `echom` 。

在分割上打草稿

接下来我们将打开一个新的分割把结果展示给用户。 这将让用户能够借助Vim的全部功能来浏览字节码，而不是仅仅只在屏幕上昙花一现。

为此我们将创建一个"草稿"分割：一个分割，它包括一个永不保存并每次执行映射都会被覆盖的缓冲区。 把 `PotionShowBytecode()` 函数改成这样：

```
:::vim
function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.

endfunction
```

新增的命令应该很好理解。

`vsplit` 创建了名为 `__Potion_Bytecode__` 的新竖直分割。 我们用下划线包起名字，使得用户注意到这不是普通的文件(它只是显示输出的缓冲区)。 下划线不是什么特殊用法，只是约定俗成罢了。

接着我们用 `normal! ggdG` 删除缓冲区中的所有东西。 第一次执行这个映射时，并不需要这样做，但之后我们将重用 `__Potion_Bytecode__` 缓冲区，所以需要清空它。

接下来我们为这个缓冲区设置两个本地设置。首先我们设置它的文件类型为 `potionbytecode` ，只是为了指明它的用途。 我们也改变 `buftype` 为 `nofile` ，告诉Vim这个缓冲区与磁盘上的文件不相关，这样它就不会把缓冲区写入。

最后还剩下把我们保存在 `bytecode` 变量的字节码转储进缓冲区。完成函数，让它看上去像这样：

```
:::vim
function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%") . " 2>&1")

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.
    call append(0, split(bytecode, '\v\n'))
endfunction
```

Vim函数 `append()` 接受两个参数：一个将被附加内容的行号和一个将按行附加的字符串列表。 举个例子，尝试执行下面命令：

```
:::vim
:call append(3, ["foo", "bar"])
```

这将附加两行， `foo` 和 `bar` ，在你当前缓冲区的第三行之后。 这次我们将在表示文件开头的第0行之后添加。

我们需要一个字符串列表来附加，但我们只有来自 `system()` 调用的单个包括换行符的字符串。 我们使用Vim的 `split()` 函数来分割这一大坨文本成一个字符串列表。 `split()` 接受一个待分割的字符串和一个查找分割点的正则表达式。这真的很简单。

现在函数已经完成了，试一下对应的映射。 当你在 `factorial.pn` 中执行 `<localleader>b` ，Vim将打开新的包括Potion字节码的缓冲区。 修改Potion源代码，保存文件，执行映射来看看会有什么不同的结果。

练习

阅读 `:help bufname` 。

阅读 `:help buftype` 。

阅读 `:help append()` 。

阅读 `:help split()` 。

阅读 `:help :!` 。

阅读 `:help :read` 和 `:help :read!` (我们没有讲到这些命令，不过它们非常好用)。

阅读 `:help system()` 。

阅读 `:help design-not` 。

目前，我们的插件要求用户在执行映射之前手动保存文件来使得他们的改变起效。 当今撤销已经变得非常轻易，所以修改写过的函数来自动替他们保存。

如果你在一个带语法错误的Potion文件上执行这个字节码映射，会发生什么？为什么？

修改 `PotionShowBytecode()` 函数来探测Potion编译器是否返回一个错误，并向用户输出错误信息。

加分题

每次你执行字节码映射时，一个新的竖直分割都会被创建，即使用户没有关闭上一个。 如果用户没有一再关闭这些窗口，他们最终将被大量额外的窗口困住。

修改 `PotionShowBytecode()` 来探测 `__Potion_Bytecode__` 缓冲区的窗口是否已经打开了， 如果是，切换到它上去而不是创建新的分割。

你大概想要阅读 `:help bufwinnr()` 来获取帮助。

额外的加分题

还记得我们设置临时缓冲区的 `filetype` 为 `potionbytecode` ？ 创建 `syntax/potionbytecode.vim` 文件并为Potion字节码定义语法高亮，使得它们更易读。

自动加载

我们已经为我们的Potion插件写了大量的功能，覆盖了本书所要讲的内容。 在结束之前，我们将讲到一些非常重要的方法，可以给我们的插件锦上添花。

第一项是使用自动加载让我们的插件更有效率。

如何自动加载

目前，当用户加载我们的插件时(比如打开了一个Potion文件)，所有的功能都会被加载。 我们的插件还很小，所以这大概不是什么大问题，但对于较大的插件，加载全部代码将会导致可被察觉的卡顿。

Vim使用称为"自动加载(autoload)"来解决这个问题。自动加载让你直到需要时才加载某一部分代码。 会有一些性能上的损失，但如果用户不总是需要你的插件的每一行代码，自动加载将带来速度上的飞跃。

示范一下它是怎么工作的。看看下面的命令：

```
:::vim
:call somefile#Hello()
```

当你执行这个命令，Vim的行为与平常的函数调用有些许不同。

如果这个函数已经加载了，Vim简单地像平常一样调用它。

否则Vim将在你的 `~/.vim` (或 `~/.vim/bundles/对应的插件/autoload`)下查找一个叫做 `autoload/somefile.vim` 的文件。

如果文件存在，Vim将加载/source文件。接着Vim就会像平常一样调用它。

在这个文件内，函数应该这样定义：

```
:::vim
function somefile#Hello()
    " ...
endfunction
```

你可以在函数名中使用多个 `#` 来表示子目录。举个例子：

```
:::vim
:call myplugin#somefile#Hello()
```

这将在 `autoload/myplugin/somefile.vim` 查找自动加载文件。 里面的函数需要使用自动加载的绝对路径进行定义：

```
:::vim
function myplugin#somefile#Hello()
    " ...
endfunction
```

实验一下

为了更好地理解自动加载，让我们实验一下。 创建一个 `~/.vim/autoload/example.vim` 文件并加入下面的内容：

```
:::vim
echom "Loading..."

function! example#Hello()
    echom "Hello, world!"
endfunction

echom "Done loading."
```

保存文件并执行 `:call example#Hello()` 。Vim将输出下面内容：

```
:::text
Loading...
Done loading.
Hello, world!
```

这个小演示证明了几件事：

1. Vim的确是在半途加载了 `example.vim` 文件。当我们打开Vim的时候它并不存在，所以不可能是在启动时加载的。
2. 当Vim找到它需要自动加载的文件后，它在调用对应函数之前就加载了整个文件。

先不要关闭**Vim**，修改函数的定义成这样：

```
:::vim
echom "Loading..."

function! example#Hello()
    echom "Hello AGAIN, world!"
endfunction

echom "Done loading."
```

保存文件并不要关闭**Vim**，执行 `:call example#Hello()` 。Vim将简单地输出：

```
:::text
Hello, world!
```

Vim已经有了 `example#Hello` 的一个定义，所以它不再需要重新加载文件，这意味着：

1. 函数以外的代码将不再执行。
2. 它不会反映函数本身的变化。

现在执行 `:call example#BadFunction()` 。你将再一次看到加载信息，伴随着一个函数不存在的错误。 但现在尝试再次执行 `:call example#Hello()` 。这次你将看到更新后的信息！

目前为止你应该清晰地了解到Vim会怎么处理一个自动加载类型的函数调用吧：

1. 它首先是否已经存在同名的函数了。如果是，就调用它。
2. 否则，查找名字对应的文件，并source它。
3. 然后试图调用那个函数。如果成功，太棒了。如果失败，就输出一个错误。

如果你还是没有完成弄懂，回到前面重新过一遍演示，注意观察每条规则生效的地方。

自动加载什么

自动加载不是没有缺陷的。 设置了自动加载后，会有一些(小的)运行开销，更别说你不得不在你的代码里容忍丑陋的函数名了。

正因为如此，如果你不是写一个用户会在每次打开Vim对话时都用到的插件，最好尽量把功能代码都挪到autoload文件中去。 这将减少你的插件在用户启动Vim时的影响，尤其是在人们安装了越来越多的插件的今天。

所以有什么是可以安全地自动加载？那些不由你的用户直接调用的部分。 映射和自定义命令不能自动加载(因为它们需要由用户调用)，但别的许多东西都可以。

让我们看看Potion插件里有什么可以自动加载的。

在Potion插件里添加自动加载

我们将从编译和执行功能开始下手。 在前一章的最后，我们的 `ftplugin/potion/running.vim` 文件大概是这样：

```
:::vim
if !exists("g:potion_command")
    let g:potion_command = "/Users/sjl/src/potion/potion"
endif

function! PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction

function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.
    call append(0, split(bytecode, '\v\n'))
endfunction

nnoremap <buffer> <localleader>r :call PotionCompileAndRunFile()<cr>
nnoremap <buffer> <localleader>b :call PotionShowBytecode()<cr>
```

这个文件仅仅当Potion文件加载时才会调用，所以它通常不会影响Vim的启动时间。 但可能会有一些用户就是不想要这些功能，所以如果我们可以自动加载某些部分， 每次打开Potion文件时可以省下他们以毫秒记的时间。

是的，这种情况下我们不会节省多少。 但你可以想象到可能有那么一个插件包括了数千行可以通过自动加载来减少每次的加载时间的代码。

让我们开始吧。在你的插件repo中创建一个 `autoload/potion/running.vim` 文件。 然后移动两个函数进去，并修改它们的名字，让它们看上去像：

```
:::vim
echom "Autoloading..."

function! potion#running#PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction

function! potion#running#PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.
    call append(0, split(bytecode, '\v\n'))
endfunction
```

注意 `potion#running` 部分的函数名怎么匹配它们所在的路径。 现在修改 `ftplugin/potion/running.vim` 文件成这样：

```
:::vim
if !exists("g:potion_command")
    let g:potion_command = "/Users/sjl/src/potion/potion"
endif

nnoremap <buffer> <localleader>r
            \ :call potion#running#PotionCompileAndRunFile()<cr>

nnoremap <buffer> <localleader>b
            \ :call potion#running#PotionShowBytecode()<cr>
```

保存文件，关闭Vim，然后打开你的 `factorial.pn` 文件。尝试这些映射，确保它们依然正常工作。

确保你仅仅在第一次执行其中一个映射的时候才看到诊断信息 `Autoloading...` (你可能需要使用 `:message` 来看到)。 一旦认为自动加载正常工作，你可以移除那些信息。

正如你看到的，我们保留 `nnoremap` 映射部分不变。 我们不能自动加载它们，不然用户就没办法引发自动加载了！

你将在Vim插件中普遍看到：大多数的功能将位于自动加载函数中，仅有 `nnoremap` 和 `command` 命令每次都被Vim加载。 每次你写一个有用的Vim插件时，不要忘了这一点。

练习

阅读`:help autoload`

稍微测试一下并弄懂自动加载变量是怎么一回事。

假设你想要强制加载一个Vim已经加载的自动加载文件，并不会惊扰到用户。 你会怎么做？你可能想要阅读`:help silent!`(译注：此处应该是`:help :silent`)。不过在现实生活中请不要那么做。

文档

我们的Potion插件有着许多有用的功能，但是无人知晓这一点，除非我们留下了文档！

Vim自身的文档非常棒。它不仅是详细地，而且也是非常透彻的。 同时，它也启发了许多插件作者写出很好的插件文档，结果是在Vimscript社区里营造出强大的文档文化。

如何使用文档

当你阅读Vim里的 `:help` 条目时，你显然注意到了有些内容被高亮得跟别的不一样。 让我们看一下这是怎么回事。

打开任何帮助条目(比如 `:help help`)并执行 `:set filetype?` 。Vim显示 `filetype=help` 。 现在执行 `:set filetype=text` ，你将看到高亮消失了。 再次执行 `:set filetype=help` ，高亮又回来了。

这表明Vim帮助文件只是获得了语法高亮的文本文件，一如其他的文件格式！ 这意味着你可以照着写并获得同样的高亮。

在你的插件repo中创建名为 `doc/potion.txt` 文件。 Vim/Pathogen在索引帮助条目时查找在 `doc` 文件夹下的文件，所以我们在此写下插件的帮助文档。

用Vim打开这个文件并执行 `:set filetype=help` ，这样你在输入的时候就可以看到语法高亮。

帮助的题头

帮助文件的格式是一个取决于个人品味的问题。 尽管这么说，我还是讲讲一种在当前的Vimscript社区逐渐被广泛使用的文档格式。

文件的第一行应该包括帮助文件的文件名，接下来是一行对插件功能的描述。 在你的 `potion.txt` 文件的第一行加上下面的内容：

```
:::text
*potion.txt* functionality for the potion programming language
```

在帮助文件中用星号括起一个词创建了一个可以用于跳转的"tag"。 执行 `:Helptags` 来让Pathogen重新索引帮助tags，接着打开一个新的Vim窗口并执行 `:help potion.txt` 。 Vim将打开你的帮助文档，一如往日打开别人写的。

接下来你应该把你的插件的大名和一个老长老长的描述打上去。 有些作者(包括我)喜欢在这上面花点心思，用一些ASCII艺术字点缀一下。 在 `potion.txt` 文件内加上一个不错的标题节：

```
:::text
*potion.txt* functionality for the potion programming language

                      ___      _   _              ~
                     / _ \___ | |_(_) ___  _ __   ~
                    / /_)/ _ \| __| |/ _ \| '_ \  ~
                   / ___/ (_) | |_| | (_) | | | | ~
                   \/    \___/ \__|_|\___/|_| |_| ~

          Functionality for the Potion programming language.
        Includes syntax highlighting, code folding, and more!
```

我是通过执行 `figlet -f ogre "Potion"` 命令来得到这些有趣的字符的。 Figlet是一个好玩的小程序，可以生成ACSII艺术字。 每一行结尾的 `~` 字符保证Vim不会尝试高亮或隐藏艺术字中的某些字符。

写什么文档

接下来通常是一个内容目录。不过，首先，让我们决定我们想要写的文档内容。

在给一个新插件写文档时，我通常从下面的列表开始：

- Introduction
- Usage
- Mappings
- Configuration
- License
- Bugs
- Contributing
- Changelog
- Credits

如果这是一个大插件，需要一个"大纲"，我将写一个介绍段落来概括它的主要功能。 否则我会跳过它继续下一段。

一个"usage"段落应该解释，大体上用户将怎样使用你的插件。如果他们需要通过映射进行交互，告诉他们。 如果映射数目不是很多，你可以简单地在这里列出来，否则你可能需要创建一个单独的"mappings"段落来一一列出。

"configuration"段落应该详尽列出用户可以自定义的变量，以及它们的功能和默认值。

"license"段落应该指出插件代码所用的协议，连同一个指向协议完整文本的URL。 不要把整份协议放入帮助文件 -- 大多数用户知道这些常用的协议是什么意思，这样做只会徒增混乱。

"bugs"段落应该尽可能短小。列出所有你已知却尚未解决的主要的bugs，并告知用户如何向你报告他们抓到的新bug。

如果你希望你的用户可以向项目奉献bug fixes和features，他们需要怎么做。 应该把pull request发到GitHub呢？还是Bitbucket?要用email寄补丁吗？ 要选择其中的一个还是全都得要？一个"contributing"段落可以清楚地表明你想要接受代码的方式。

一个changlog也是值得包含在内的，这样当用户从版本X更新到版本Y时，他们立即可以看到什么改变了。 此外，我强烈推荐你为你的插件使用一个合理的版本号计划，比如Semantic Versioning，并一直坚持。 你的用户会感谢你的。

最后，我喜欢加入一个"credits"段落来留下自己的大名，列出影响该插件创作的其他插件，感谢奉献者，等等。

这样我们就有一个成功的开始了。对于许多特殊的插件，你可能觉得需要不这么做，那也没问题。 你仅需追随下面几条规则：

- 透彻明了
- 不要废话
- 带领你的用户漫步于你的插件，从一无所知到了如指掌。

内容目录

既然我们已经了解了应该要有的段落，把下面内容加入到 `potion.txt` 文件中：

```
:::text
====================================================================
CONTENTS                                            *PotionContents*

    1. Usage ................ |PotionUsage|
    2. Mappings ............. |PotionMappings|
    3. License .............. |PotionLicense|
    4. Bugs ................. |PotionBugs|
    5. Contributing ......... |PotionContributing|
    6. Changelog ............ |PotionChangelog|
    7. Credits .............. |PotionCredits|
```

有很多事情需要在这里提一下。 首先， `=` 字符组成的那行将被高亮。你可以用这些行醒目地隔开你的帮助文档各部分。 你也可以使用 `-` 隔开子段落，如果想要的话。

`*PotionContents*` 将创建另一个tag，这样用户可以输入 `:help PotionContents` 来直接跳到内容目录。

用 `|` 包起每一个词将创建一个跳转到tag的链接。 用户可以把他们的光标移到词上并按下 `<c-]>` 跳转到对应tag，就像他们输入 `:help TheTag` 一样。 他们也可以用鼠标点开。

Vim将隐藏 `*` 和 `|` 并高亮其间的内容，所以用户将会看到一个整洁漂亮的目录，以便于跳到感兴趣的地方。

段落

你可以像这样创建一个段头：

```
:::text
====================================================================
Section 1: Usage                                       *PotionUsage*

This plugin with automatically provide syntax highlighting for
Potion files (files ending in .pn).

It also...
```

确保对 `*` 围起的内容创建了正确的tags，这样你的目录的链接才能正常工作。

继续并为目录中每一部分创建段头。

例子

我可以讲述所有的帮助文件语法和怎样使用它们，但我不喜欢这样。 所以，不如我给你一系列不同类型的Vim插件文档作为例子。

对于每个例子，复制文档源代码到一个Vim缓冲区并设置它的filetype为 `help` 来观察它的渲染。 如果你想比较每个渲染效果，切回 `text` 看看。

你也许需要使用你的Vimscript技能为当前缓冲区创建一个切换于 `help` 和 `text` 的映射。

- Clam，我自己用来写shell命令的插件。这是一个很小的范例，满足了我前面讲过的大多数内容。
- NERD tree，Scrooloose写的一个文件浏览插件。 注意大体结构，还有他如何在详尽解释每一项之前，总结出一个易读的列表。
- Surround，Tim Pope写的一个处理环绕字符的插件。 注意到它没有目录，以及不同的段头风格和表格列项(table column headers)。 弄懂他是怎么做到的，并想想你是否喜欢这种风格。这是个人风格问题啦。
- Splice，我自己用来解决版本控制中three-way merge conflict的插件。 注意映射列表的排版方式，以及我怎样使用ASCII流派的图片来解释布局。有时候，一图胜千言。

不要忘了，Vim本身的文档也可以作为一个例子。这会给你许多可供学习的内容。

写！

既然你已经看过其他插件如何规划和撰写它们的文档，给你的Potion插件填补上文档内容吧。

如果你不熟悉技术文档的写作，这可能会是个挑战。 学习如何去写并不容易，但一如其他技能，它需要的是更多的练习，所以现在开始吧！ 你不必苛求完美，从战争中学习战争即可。

不要惧于写你没有完全弄懂的事情，待会丢掉它重写即可。 经常只要在缓冲区中信手留下几笔，将会带动你的头脑进入写作状态。 任何时候你想重起炉灶，旧版本一直会在版本控制中等你。

一个开始的好方法是想象你身边也有一个使用Vim的好基友。 他对你的插件很感兴趣却未曾用过，而你的目标是让他熟练掌握。 在你写下插件工作的方式之前，考虑如何向人类进行介绍，可以让你脚踏实地，避免太深入于技术层面。

如果你依旧卡住了，感觉自己无力应对写一个完整插件的文档的挑战，尝试做点简单的。 在你的 `~/.vimrc` 中找一个映射并给它写下完整的文档。解释它是干什么的，怎么用它，它怎么工作。 比如，这是我的 `~/.vimrc` 的一个例子：

```
:::vim
" "Uppercase word" mapping.
"
" This mapping allows you to press <c-u> in insert mode to convert the
" current word to uppercase.  It's handy when you're writing names of
" constants and don't want to use Capslock.
"
" To use it you type the name of the constant in lowercase.  While
" your cursor is at the end of the word, press <c-u> to uppercase it,
" and then continue happily on your way:
"
"                            cursor
"                            v
"     max_connections_allowed|
"     <c-u>
"     MAX_CONNECTIONS_ALLOWED|
"                            ^
"                            cursor
"
" It works by exiting out of insert mode, recording the current cursor
" location in the z mark, using gUiw to uppercase inside the current
" word, moving back to the z mark, and entering insert mode again.
"
" Note that this will overwrite the contents of the z mark.  I never
" use it, but if you do you'll probably want to use another mark.
inoremap <C-u> <esc>mzgUiw`za
```

它比一个完整插件的文档短很多，却是一个练习写作的好练习。 如果你把 `~/.vimrc` 放到Bitbucket或GitHub， 别人也更容易理解。

练习

给Potion插件每一部分写下文档。

阅读 `:help help-writing` 来帮助你写帮助文档。

阅读 `:help :left` ， `:help :right` ,和 `:help :center` 来学习三个有用的命令使得你的ASCII艺术字更好看。

发布

现在你拥有了足够的Vimscript技能来打造能帮助许多人的Vim插件。 这一章涉及如何把你的插件发布在网上，以便人们获取，还有如何向潜在用户派小广告。

托管

你需要做的第一件事是把你的插件放在网上，让其他人可以下载它。 最普遍的选择是放到Vim官网的script版面。

你需要这个网站的一个免费账户。一旦你有了，你可以点击"Add Script"链接并填写表单。 到那里你就会明白了。

在过去的几年中有一个趋势，越来越多的插件托管在类似Bitbucket或GitHub的网络集市上。这种情况可能由于两个因素。首先，Pathogen使得每一个被安装的插件的文件不需要放在单独的位置。 像Mercurial和Git这样的分布式版本控制系统以及像Bitbucket和GitHub这样的公共托管网站的崛起对此也有影响。

提供代码仓库对于想要用版本控制管理自己的dotfiles的人来说是十分方便的。 Mercurial用户可以使用Mercurial的"subrepositories"来跟踪插件版本的变化， 而Git用户可以使用submodules(尽管只能对其他Git代码仓库起作用，这跟Mercurial的subrepo不一样)。

对你安装的每一个插件有一个完整的仓库，也使得当发现它们出现问题时debug更简单。 你可以使用blame, bisection或其他你的VCS提供的工具来找出哪里的问题。 如果你在自己的机器上有一个仓库，奉献fixes也会变得更简单。

希望你已经决定把你的插件代码仓库公开出来。无论你采用了哪家的服务，至少代码库需要能够被人们获取。

文档

你已经用Vim自己的帮助文档格式透彻地给插件作了文档。但你的工作还没完成呢。 你还需要写出一个简介，包括下面几条：

1. 你的插件可以用来干什么？
2. 为什么用户想要用它？
3. 为什么它比同类的插件(如果有的话)要好？
4. 它遵循什么协议？
5. 一个到完整文档的链接，可以考虑借助vim-doc网站进行渲染。

这些应该放在你的README文件(它将会显示在Bitbucket或GitHub的版本库的主页面)，你也可以把它作为Vim.org上的插件描述。

包括一些屏幕截图总是一个好主意。作为一个文本编辑器不意味着Vim没有一个用户界面。

贴小广告

一旦你已经把插件部署到各个托管网站上，是时候向全世界宣传它的到来！ 你可以在Twitter上向你的粉丝介绍，在Reddit的/r/vim版面推广它，在你的个人网站上写关于它的博客， 在Vim邮件列表上给新手们派小广告。

无论何时，当你推出自己创作的东西，你总会收到一些赞美和批评。 不要对不好的评价耿耿于怀。倾听他们的呼声，同时厚着脸皮，心态平和地对待作品中被指出的小瑕缺(不管对还是不对)。 没有什么是十全十美的，而且这就是Internet，所以如果你想保持快乐和激情，你需要拿得起放得下。

练习

如果你还没有Vim.org账户，创建一个。

察看你喜欢的插件的READEME文件，看看它们是怎么组织起来的以及它们包含的信息。

还剩下什么？

如果已经读到了这里并且完成了所有的例子和练习，你现在对Vimscript基础的掌握就很牢固了。 不要担心，还有许多东西需要学呢！

如果你求知若渴，这里还有一些东西值得你去探索。

配色方案

在本书中我们给Potion文件添加了语法高亮。作为硬币的另一面，我们也可以创建配色方案来决定每种语法元素的颜色。

制作Vim的配色方案非常简单直白，甚至有点重复。阅读 `:help highlgiht` 来学习基础知识。你可能想要看看一些内置的配色方案来看他们怎么组织文件的。

如果你渴望挑战，看看我自己的灰太狼配色方案来了解我是怎么用Vimscript来为我简化定义及维护工作的。 注意"palette"字典和 `HL` 函数，它们动态地生成 `highlight` 命令。

Command命令

许多插件允许用户使用键映射和函数调用来交互，但有一些偏好使用Ex命令。 举个例子，Fugitive插件创建类似 `:Gbrowse` 和 `:Gdiff` 并把调用它们的方式留给用户定制。

像这样的命令是通过 `:command` 命令创建的。阅读 `:help user-commands` 来学习怎样给自己制作一个。 你应该已经学会了足够的Vimscript知识来帮助自己理解Vim文档，并以此来学习新的命令。

运行时路径

在本书中，关于Vim怎么加载某个文件时，我都是用"使用Pathogen"应付过去的。 鉴于你已经懂得了许多Vimscript知识，你可以阅读 `:help runtimepath` 并查看Pathogen源代码 来找出幕后隐藏的真相。

Omnicomplete

Vim提供了许多不同的方法来补全文本(浏览 `:help ins-completion`)。 大多数都很简单，但其中最强大的是"omnicomplete"， 它允许你调用一个自定义的Vimscript函数来决定你想到的各种补全方式。

当你决定对omnicomplete一探究竟，你可以从 `:help omnifunc` 和 `:help coml-omni` 开始你的征途。

编译器支持

在我们的Potion插件中，我们创建了一些编译并执行Potion文件的映射。 Vim提供了更深入的支持来跟编译器交互，包括解析编译器错误并生成一个整洁的列表让你跳转到对应的错误。

如果你对此感兴趣，你可以从通读整篇 `:help quickfix.txt` 开始深入。 不过，我得提醒你 `errorformat` 不适合心脏虚弱的人阅读。

其他语言

这本书专注于Vimscript，但Vim也提供了其他语言的接口，比如Python, Ruby, 和Lua。 这意味着如果不喜欢Vimscript，你可以使用其他语言拓展Vim。

当然还是需要了解Vimscript来编辑你的 `~/.vimrc` ，和理解Vim提供给其他语言的API。 但使用一个替代语言可能是从Vimscript的局限之处解放出来的好办法，尤其在写大型插件的时候。

如果你想了解更多用特定语言拓展Vim，查看下列对应的帮助文档：

- `:help Python`
- `:help Ruby`
- `:help Lua`
- `:help perl-using`
- `:help MzScheme`

Vim文档

作为最后的部分，这里列出了一些Vim帮助条目，它们非常有用，有趣，有道理，或者仅仅是好玩(排名不分先后)：

- `:help various-motions`
- `:help sign-support`
- `:help virtualedit`
- `:help map-alt-keys`

- `:help error-messages`
- `:help development`
- `:help tips`
- `:help 24.8`
- `:help 24.9`
- `:help usr_12.txt`
- `:help usr_26.txt`
- `:help usr_32.txt`
- `:help usr_42.txt`

练习

去为你想要的功能写一个Vim插件，向全世界分享你的成果！